Agilent Standard Instrument Conrol Library User's Guide for Windows

Agilent Technologies

Contents

Agilent Standard Instrument Control Library User's Guide for Windows

Notice

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

Agilent Technologies shall not be liable for any errors contained in this document. *Agilent Technologies makes no warranties of any kind with regard to this document, whether express or implied. Agilent Technologies specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.* Agilent Technologies 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 Agilent Technologies product and replacement parts can be obtained from Agilent Technologies, Inc.

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 Agilent standard software agreement for the product involved.

Trademark Information

Printing History

Edition 1 - April 1994
Edition 2 - September 1995
Edition 3 - May 1996
Edition 4 - October 1996
Edition 5 - July 2000

Copyright Information

1

Introduction

Introduction

This *Agilent Standard Instrument Control Libraries (SICL) User's Guide for Windows* describes Agilent SICL and how to use it to develop I/O applications on Microsoft Windows 95, Windows 98, Windows NT 4.0, and Windows 2000. A getting started chapter is provided to help you write and run your first SICL program. Then, this guide explains how to build and program SICL applications. Later chapters are interface-specific, describing how to use SICL with GPIB, GPIO, VXI, RS-232, and LAN interfaces.

NOTE

Before you can use SICL, you must install and configure SICL on your computer. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for installation on Windows systems. Unless otherwise indicated, Windows NT refers to Windows NT 4.0.

If You Need Help:

- In the USA and Canada, you can reach Agilent Technologies at these telephone numbers:

  USA: 1-800-452-4844
  Canada: 1-877-894-4414

- Outside the USA and Canada, contact your country's Agilent support organization. A list of contact information for other countries is available on the Agilent web site:

  http://www.agilent.com/find/assist

What's in This Guide?

This chapter provides an overview of SICL. In addition, this guide contains the following chapters:

- *Chapter 2 - Getting Started with SICL* shows how to build and run an example program in C/C++ and Visual BASIC.
- *Chapter 3 - Programming with SICL* shows how to build a SICL application in a Windows environment and provides information on communications sessions, addressing, error handling, locking, etc..
- *Chapter 4 - Using SICL with GPIB* shows how to communicate over the GPIB interface.
- *Chapter 5 - Using SICL with GPIO* shows how to communicate over the GPIO interface.
- *Chapter 6 - Using SICL with VXI* shows how to communicate over the VXIbus interface.
- *Chapter 7 - Using SICL with RS-232* shows how to communicate over the RS-232 interface.
- *Chapter 8 - Using SICL with LAN* shows how to communicate over a Local Area Network (LAN).
- *Chapter 9 - Troubleshooting SICL Programs* describes some common SICL programming problems and provides troubleshooting procedures.
- *Chapter 10 - More SICL Example Programs* contains additional example programs to help you develop SICL applications.
- *Chapter 11 - SICL Language Reference* provides function syntax and description for each SICL function.
- *Appendix A - SICL System Information* provides information on SICL software files and system interaction.

- *Appendix B - Porting to Visual Basic* explains how to move SICL applications from earlier versions of Visual Basic (such as version 3.0) to Visual Basic version 4.0 and above.
- *Appendix C - SICL Error Codes* provides a list of error codes and error strings along with a brief description of each error.
- *Appendix D - SICL Function Summary* summarizes supported features for each SICL function.
- *Appendix E - RS-232 Cables* lists part numbers and shows wiring diagrams for several RS-232 cables.
- *Glossary* includes major terms and definitions used in this guide.

SICL Overview

SICL is part of the Agilent IO Libraries. The Agilent IO Libraries consists of two libraries: *Agilent Virtual Instrument Software Architecture (VISA)* and *Agilent Standard Instrument Control Library (SICL).*

Introducing VISA and SICL

- Agilent Virtual Instrument Software Architecture (VISA) is an I/O library designed according to the VXI*plug&play* System Alliance that allows software developed from different vendors to run on the same system.
- Use VISA if you want to use VXI*plug&play* instrument drivers in your applications, or if you want the I/O applications or instrument drivers that you develop to be compliant with VXI*plug&play* standards. If you are using new instruments or are developing new I/O applications or instrument drivers, we recommend you use Agilent VISA.
- Agilent Standard Instrument Control Library (SICL) is an I/O library developed by Hewlett-Packard and Agilent that is portable across many I/O interfaces and systems.
- You can use Agilent SICL if you have been using SICL and want to remain compatible with software currently implemented in SICL.

NOTE

Since VISA and SICL are different libraries, using VISA functions and SICL functions in the same I/O application is not supported.

SICL Description

Agilent Standard Instrument Control Library (SICL) is an I/O library developed by Hewlett-Packard and Agilent that is portable across many I/O interfaces and systems. 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 with no (or 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.

SICL Support

The 32-bit version of SICL is supported on this version of the Agilent IO Libraries for Windows 95, Windows 98, Windows NT, and Windows 2000. Support for the 16-bit version of SICL was removed in version H.01.00. However, versions through G.02.02 support 16-bit SICL. C, C++, and Visual BASIC are supported on all these Windows versions. SICL is supported on the GPIB, GPIO, VXI, RS-232, and LAN interfaces.

SICL Users

SICL is intended for instrument I/O and C/C++ or Visual BASIC programmers who are familiar with Windows 95, Windows 98, Windows 2000, or Windows NT. To perform SICL installation and configuration on Windows NT, you must have system administrator privileges on the Windows NT system.

SICL Documentation

This table shows associated documentation you can use when programming with Agilent SICL.

Agilent SICL Documentation

Document	Description
Agilent SICL User's Guide for Windows	Shows how to use Agilent SICL and provides the SICL language reference.
SICL Online Help	Information is provided in the form of Windows Help.
SICL Example Programs	Example programs are provided online to help you develop SICL applications. SICL example programs are provided in the **`C\SAMPLES`** (for C/C++) subdirectory and in the **`VB\SAMPLES`** subdirectory (for Visual BASIC) under the base directory where SICL is installed. For example, under the **`C:\SICL95`** or **`C:\SICLNT`** base directory if the default installation directory was used.
VXIbus Consortium specifications (when using VISA 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

Getting Started with SICL

Getting Started with SICL

This chapter gives guidelines to help you to get started programming with SICL using the C/C++ language. This chapter provides example programs in C/C++ and in Visual Basic to help you verify your configuration and introduce you to some of SICL's basic features. The chapter contents are:

- Getting Started Using C
- Getting Started Using Visual Basic

NOTE

This chapter is divided into two sections: the first section is for C programmers and the second section is for Visual BASIC programmers. See "Getting Started Using C" if you want to use SICL with the C/C++ programming language. See "Getting Started Using Visual Basic" if you want to use SICL with the Visual BASIC programming language.

You may want to see *Chapter 11 - SICL Language Reference* to familiarize yourself with SICL functions. This reference information is also available as online help. To see the reference information online, double-click the Help icon in the **`Agilent IO Libraries`** program group.

Getting Started Using C

This section describes an example program called **IDN** that queries a GPIB instrument for its identification string. This example builds a console application for WIN32 programs (32-bit SICL programs on Windows 95, Windows 98, Windows 2000, or Windows NT) using the C programming language.

C Program Example Code

All files used to develop SICL applications in C or C++ are located in the C subdirectory of the base **IO Libraries** directory. Sample C/C++ programs are located in the **C\SAMPLES** subdirectory of the base **IO Libraries** directory.

Each sample program subdirectory contains makefiles or project files that you can use to build each sample C program. You must first compile the sample C/C++ programs before you can execute them.

The **IDN** example files are located in the **C\SAMPLES\IDN** subdirectory under the base **IO Libraries** directory. This subdirectory contains the source program, **IDN.C**. The source file **IDN.C** is listed on the following pages. An explanation of the function calls in the example follows the program listing.

```
/* This program uses the Standard Instrument Control Library to
   query a GPIB instrument for an identification string and then
   prints the result. This program is to be built as a WIN32 console
   application on Windows 95, Windows 98, Windows 2000, or Windows NT.
   Edit the DEVICE_ADDRESS line to specify the address of the applicable
   device. For example:

   hpib7,0: refers to a GPIB device at bus address 0 connected to
            an interface named “hpib7” by the IO Config utility.

   hpib7,9,0: refers to a GPIB device at bus address 9, secondary
              address 0, connected to an interface named “hpib7”
              by the IO Config utility. */

#include <stdio.h>    /* for printf() */
#include “sicl.h”     /* SICL routines */
#define DEVICE_ADDRESS “hpib7,0” /* Modify to match your setup */
```

```
void main(void)
{
INST id;                  /* device session id */
char buf[256] = { 0 };    /* read buffer for idn string */

#if defined(__BORLANDC__) && !defined(__WIN32__)
    _InitEasyWin();// required for Borland EasyWin programs.
#endif

/* Install a default SICL error handler that logs an error message and exits.
   On Windows 95 or Windows 98, view messages with the SICL Message Viewer,
   and on Windows 2000 or Windows NT use the Event Viewer. */
ionerror(I_ERROR_EXIT);

/* Open a device session using the DEVICE_ADDRESS */
id = iopen(DEVICE_ADDRESS);

/* Set the I/O timeout value for this session to 1 second */
itimeout(id, 1000);

/* Write the *RST string (and send an EOI indicator) to put the instrument
   into a known state. */
iprintf(id, "*RST\n");

/* Write the *IDN? string and send an EOI indicator, then read the response
   into buf.
ipromptf(id, "*IDN?\n", "%t", buf);
printf("%s\n", buf);
iclose(id);

/* This call is a no-op for WIN32 programs.*/
_siclcleanup();
}
```

C Example Code Description

sicl.h. The **`sicl.h`** file is included at the beginning of the file to provide the function prototypes and constants defined by SICL.

INST. Notice the declaration of **`INST`** *id* at the beginning of **`main`**. The type **`INST`** is defined by SICL and is used to represent a unique identifier that will describe the specific device or interface that you are communicating with. The *id* is set by the return value of the SICL **`iopen`** call and will be set to 0 if **`iopen`** fails for any reason.

ionerror. The first SICL call, **`ionerror`**, installs a default error handling routine that is automatically called if any of the subsequent SICL calls result in an error. **`I_ERROR_EXIT`** specifies a built-in error handler that will print out a message about the error and then exit the program. If desired, a custom error handling routine could be specified instead.

NOTE

On Windows 95, Windows 98, and Windows 2000, error messages may be viewed by executing the **`Message Viewer`** utility in the **`Agilent IO Libraries`** program group. On Windows NT, these messages may be viewed with the **`Event Viewer`** utility in the Agilent **`IO Libraries Control`** on the taskbar.

iopen. When an **`iopen`** call is made, the parameter string **`"hpib7,0"`** passed to **`iopen`** specifies the GPIB interface followed by the bus address of the instrument. The interface name, **`"hpib7"`**, is the name given to the interface during execution of the **`IO Config`** utility. The bus (primary) address of the instrument follows (**`"0"`** in this case) and is typically set with switches on the instrument or from the front panel of the instrument.

NOTE

To modify the program to set the interface name and instrument address to those applicable for your setup, see *Chapter 3 - Programming with SICL* for information on using SICL's addressing capabilities.

itimeout. **`itimeout`** is called to set the length of time (in milliseconds) that SICL will wait for an instrument to respond. The specified value will depend on the needs of your configuration. Different timeout values can be set for different sessions as needed.

iprintf and ipromptf. SICL provides formatted I/O functions that are patterned after those used in the C programming language. These SICL functions support the standard ANSI C format strings, plus additional formats defined specifically for instrument I/O.

The SICL **`iprintf`** call sends the Standard Commands for Programmable Instruments (SCPI) command **`*RST`** to the instrument that puts it in a known state. Then, **`ipromptf`** queries the instrument for its identification string. The string is read back into **`buf`** and then printed to the screen. (Separate **`iprintf`** and **`iscanf`** calls could have been used to perform this operation.)

The **`%t`** read format string specifies that an ASCII string is to be read back, with end indicator termination. SICL automatically handles all addressing and GPIB bus management necessary to perform these reads and writes to instrument.

iclose and _siclcleanup. The **`iclose`** function closes the device session to this instrument (*id* is no longer valid after this point). WIN32 programs on Windows 95, Windows 98, Windows 2000, or Windows NT do not require the **`_siclcleanup`** call.

NOTE

See *Chapter 11 - SICL Language Reference* or the SICL online **`Help`** for more information on these SICL function calls.

Compiling the C Example Program

The **C\SAMPLES\IDN** subdirectory contains a number of files you can use to build the example with specific compilers. You will have a subset of the following files, depending on the Windows environment you are using.

IDN.C	Example program source file.
IDN.DEF	Module definition file for the IDN example program.
MSCIDN.MAK	Windows 3.1 makefile for Microsoft C and Microsoft SDK compilers.
VCIDN.MAK	Windows 3.1 project file for Microsoft Visual C++.
VCIDN32.MAK	Windows 95 or Windows NT (32-bit) project file for Microsoft Visual C++.
VCIDN16.MAK	Windows 95 (16-bit) project file for Microsoft Visual C++.
QCIDN.MAK	Windows 3.1 project file for Microsoft QuickC for Windows.
BCIDN.IDE	Windows 3.1 project file for Borland C Integrated Development Environment.
BCIDN32.IDE	Windows 95 or Windows NT (32-bit) project file for Borland C Integrated Development Environment.
BCIDN16.IDE	Windows 95 (16-bit) project file for Borland C Integrated Development Environment.

Steps to compile the **IDN** example program follow.

1. Connect an instrument to a GPIB interface that is compatible with IEEE 488.2.

2. Change directories to the location of the example.

3. The program assumes the GPIB interface name is **hpib7** (set using **IO Config**) and the instrument is at bus address **0**. If necessary, modify the interface name and instrument address on the **DEVICE_ADDRESS** definition line in the **IDN.C** source file.

4. Select and load the appropriate project or make file. Then, compile the program as follows:

   - For Borland compilers, use **`Project | Open Project`**. Then, select **`Project | Build All`**.

   - For Microsoft compilers, use **`Project | Open`**. Next, set the include file path by selecting **`Options | Directories`**. Then, in the Include File Path box, enter the full path to the **`C`** subdirectory. Finally, select **`Project | Re-build All`**.

Running the C Example Program

To run the **`IDN`** example program, execute the program from a console command prompt.

- For Borland, select **`Run | Run`**
- For Microsoft, select **`Project | Execute`** or **`Run | Go`**

If the program runs correctly, an example of the output if connected to a 54601A oscilloscope is

```
HEWLETT-PACKARD,54601A,0,1.7
```

If the program does not run, see the message logger for a list of run-time errors, and see *Chapter 9 - Troubleshooting SICL Programs* for guidelines to correct the problem.

Where to Go Next

Go to *Chapter 3 - Programming with SICL*. In addition, you should see the chapter(s) that describe how to use SICL with your specific interface(s):

- *Chapter 4 - Using SICL with GPIB*
- *Chapter 5 - Using SICL with GPIO*
- *Chapter 6 - Using SICL with VXI*
- *Chapter 7 - Using SICL with RS-232*
- *Chapter 8 - Using SICL with LAN*

You may also want to familiarize yourself with SICL functions, defined in *Chapter 11 - SICL Language Reference* and in the reference information provided in SICL online **`Help`**. If you have any problems, see *Chapter 9 - Troubleshooting SICL Programs*.

Getting Started Using Visual Basic

There is a collection of Visual Basic sample programs in the **VB\SAMPLES** subdirectory of the base **IO Libraries** directory. See these programs as examples of using SICL with Visual Basic.

Be sure to include the sicl4.bas file (in the **VB** directory) in your Visual Basic project. This file contains the necessary SICL definitions, function prototypes, and support procedures to allow you to call SICL functions from Visual Basic.

Where to Go Next

Go to *Chapter 3 - Programming with SICL*. In addition, you should see the chapter(s) that describe how to use SICL with your specific interface(s):

- *Chapter 4 - Using SICL with GPIB*
- *Chapter 5 - Using SICL with GPIO*
- *Chapter 6 - Using SICL with VXI*
- *Chapter 7 - Using SICL with RS-232*
- *Chapter 8 - Using SICL with LAN*

You may also want to familiarize yourself with SICL functions, defined in *Chapter 11 - SICL Language Reference* and in the reference information provided in SICL online **Help**. If you have any problems, see *Chapter 9 - Troubleshooting SICL Programs*.

Notes:

3

Programming with SICL

Programming with SICL

This chapter describes how to build a SICL application and then describes SICL programming techniques. Example programs are also provided to help you develop SICL applications. The chapter includes:

- Building a SICL Application
- Opening a Communications Session
- Sending I/O Commands
- Handling Asynchronous Events
- Handling Errors
- Using Locks

NOTE

Copies of the example programs are located in the **`C\SAMPLES\MISC`** subdirectory (for C/C++) or in the **`VB\SAMPLES\MISC`** subdirectory (for Visual Basic) under the base **`IO Libraries`** directory. For details on SICL functions, see *Chapter 11 - SICL Language Reference* or SICL online **`Help`**.

Building a SICL Application

This section gives guidelines to build a SICL application in a Windows environment.

Including the SICL Declaration File

For C and C++ programs, you must include the **sicl.h** header file at the beginning of every file that contains SICL function calls. This header file contains the SICL function prototypes and the definitions for all SICL constants and error codes.

```
#include "sicl.h"
```

For Visual Basic version 3.0 or earlier programs, you must add the **SICL.BAS** file to each project that calls SICL. For Visual Basic version 4.0 or later programs, you must add the **SICL4.BAS** file to each project that calls SICL.

Libraries for C Applications and DLLs

All WIN32 applications and DLLs that use SICL must link to the **SICL32.LIB** import library. (Borland compilers use **BCSICL32.DLL**.)

The SICL libraries are located in the **C** directory under the **IO Libraries** base directory (for example, **C:\Program Files\Agilent\IO Libraries\C** if you installed SICL in the default location). You may want to add this directory to the library file path used by your language tools.

Use the DLL version of the C run-time libraries, because the run-time libraries contain global variables that must be shared between your application and the SICL DLL.

If you use the static version of the C run-time libraries, these global variables will not be shared and unpredictable results could occur. For example, if you use **isscanf** with the **%F** format, an application error will occur. The following sections describe how to use the DLL versions of the run-time libraries.

Compiling and Linking C Applications

A summary of important compiler-specific considerations follows for several C/C++ compiler products when developing WIN32 applications.

> **NOTE**
>
> If you are using a version of the Microsoft or Borland compilers other than those listed in this subsection, the menu structure and selections may be different than indicated here. However, the equivalent functionality exists for your specific version.

Microsoft Visual C++ Compilers

1. Select **`Project | Settings`** or **`Build | Settings`** from the menu (depending on the version of your compiler).

2. Click the **`C/C++`** button. Then, select **`Code Generation`** from the **`Category`** list box and select **`Multithreaded Using DLL`** from the **`Use Run-Time Library`** list box. Click **`OK`** to close the dialog box.

3. Select **`Project | Settings`** or **`Build | Settings`** from the menu. Click the **`Link`** button. Then, add **`sicl32.lib`** to the **`Object/Library Modules`** list box. Click **`OK`** to close the dialog box.

4. You may want to add the SICL **`C`** directory (for example, **`C:\Program Files\Agilent\IO Libraries\C`** to the include file and library file search paths. To do this, select **`Tools | Options`** from the menu and click the **`Directories`** button. Then:

   - To set the include file path, select **`Include Files`** from the **`Show Directories for:`** list box. Next, click the **`Add`** button and type in **`C:\Program Files\Agilent\IO Libraries\C`**. Then, click **`OK`**.

   - To set the library file path, select **`Library Files`** from the **`Show Directories for:`** list box. Next, click the **`Add`** button and type in **`C:\Program Files\Agilent\IO Libraries\C`**. Then, click **`OK`**.

Borland C++ Version 4.0 Compilers

1. Link your programs with **BCSICL32.LIB**, *not* **SICL32.LIB**. **BCSICL32.LIB** is located in the **C** subdirectory under the SICL base directory (for example, **C:\Program Files\Agilent\IO Libraries\C** if SICL is installed in the default location).

2. Edit the **BCC32.CFG** and **TLINK32.CFG** files, which are located in the **BIN** subdirectory of the Borland C installation directory.

   - Add the following line to **BCC32.CFG** so the compiler can find the **sicl.h** file:

     `-IC:\`*IO_base_dir*`\C`

     where *IO_base_dir* is the **IO Libraries** base directory.

   - Add the following line to both files so the compiler and linker can find **BCSICL32.LIB**:

     `-LC:\`*IO_base_dir*`\C`

     where *IO_base_dir* is the **IO Libraries** base directory.

   - For example, to create **MYPROG.EXE** from **MYPROG.C**, type:

     ```
     BCC32 MYPROG.C BCSICL32.LIB
     ```

Loading and Running Visual Basic Applications

To load and run an existing Visual Basic application, first run Visual Basic. Then, open the project file for the program you want to run by selecting **File | Open Project** from the Visual Basic menu. Visual Basic project files have a **.MAK** file extension. After you haved opened the application's project file, you can run the application by pressing **F5** or the **Run** button on the Visual Basic Toolbar.

You can create a standalone executable (**.EXE**) version of this program by selecting **File | Make EXE File** from the Visual Basic menu. Once this is done, the application can be run stand-alone (just like any other **.EXE** file) without having to run Visual Basic.

Thread Support for 32-bit Windows Applications

SICL can be used in multi-threaded designs and SICL calls can be made from multiple threads, in WIN32 applications. However, there are some important points:

- SICL error handlers (installed with **`ionerror`**) are *per process* (not per thread) but are called in the context of the thread that caused the error to occur. Calling **`ionerror`** from one thread will overwrite any error handler presently installed by another thread.
- The **`igeterrno`** is per thread and returns the last SICL error that occurred in the current thread.
- You may want to make use of the SICL session locking functions (**`ilock`** and **`iunlock`**) to help coordinate common instrument accesses from more than one thread.
- See *Chapter 8 - Using SICL with LAN* for thread information when using SICL with LAN.

Opening a Communications Session

A communications session is a channel of communication with a particular device, interface, or commander:

- A **device session** is used to communicate with a device on an interface. A device is a unit that receives commands from a controller. Typically a device is an instrument but could be a computer, a plotter, or a printer.
- An **interface session** is used to communicate with a specified interface. Interface sessions allow you to use interface-specific functions (for example, **`igpibsendcmd`**).
- A **commander session** is used to communicate with the interface's commander. Typically a commander session is used when a computer is acting like a device.

Steps to Open a Communications Session

There are two parts to opening a communications session with a specific device, interface, or commander. First, you must declare a variable for the SICL session identifier. C and C++ programs should declare the session variable to be of type **`INST`**. Visual Basic programs should declare the session variable to be of type **`Integer`**. Once the variable is declared, you can open the communication channel by using the SICL **`iopen`** function, as shown in the following example.

C example:

```
INST id;
id = iopen (addr);
```

Visual Basic example:

```
Dim id As Integer
id = iopen (addr)
```

Where *id* is the session identifier used to communicate to a device, interface, or commander. The *addr* parameter specifies a device or interface address, or the term **`cmdr`** for a commander session. See the sections that follow for details on creating the different types of communications sessions.

Your program may have several sessions open at the same time by creating multiple session identifiers with the **iopen** function. Use the SICL **iclose** function to close a channel of communication.

Device Sessions

A **device session** allows you direct access to a device without knowing the type of interface to which the device is connected. On GPIB, for example, you do not have to address a device to listen before sending data to it. This insulation makes applications more robust and portable across interfaces, and is recommended for most applications.

Device sessions are the recommended way of communicating using SICL. They provide the highest level programming method, best overall performance, and best portability.

Addressing Device Sessions

To create a device session, specify the interface logical unit or symbolic name and a specific device logical address in the *addr* parameter of the **iopen** function. The logical unit is an integer corresponding to the interface.

The device address generally consists of an integer that corresponds to the device's bus address. It may also include a secondary address which is an integer. (Secondary addressing is not supported on RS-232 interfaces.) The following are valid device addresses:

7,23	Device at address 23 connected to an interface card at logical unit 7.
7,23,1	Device at address 23, secondary address 1, connected to an interface card at logical unit 7.
hpib,23	GPIB device at address 23.
hpib2,23,1	GPIB device at address 23, secondary address 1, connected to a second GPIB interface card.
com1,488	RS-232 device

The interface logical unit and symbolic name are set by running the **IO Config** utility from the **Agilent IO Libraries Control** (on the taskbar) for Windows 95, Windows 98, Windows 2000, or Windows NT. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on the **IO Config** utility.

Examples: Opening a Device Session

The following examples open a device session with a GPIB device at address 23.

C example:

```
INST dmm;
dmm = iopen (“hpib,23”);
```

Visual Basic example:

```
Dim dmm As Integer
dmm = iopen (“hpib,23”)
```

Interface Sessions

An **interface session** allows direct, low-level control of the specified interface. A full set of interface-specific SICL functions existds for programming features that are specific to a particular interface type (GPIB, Serial, etc.). This provides full control of the activities on a given interface, but does create less portable code.

Addressing Interface Sessions

To create an interface session, specify the particular interface logical unit or symbolic name in the *addr* parameter of the **`iopen`** function. The interface logical unit and symbolic name are set by running the **`IO Config`** utility from the **`Agilent IO Libraries Control`** (on the taskbar) for Windows 95, Windows 98, Windows 2000, or Windows NT. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on the **`IO Config`** utility.

The logical unit is an integer that corresponds to a specific interface. The symbolic name is a string which uniquely describes the interface. The following are valid interface addresses:

`7`	Interface card at logical unit 7
`hpib`	GPIB interface card.
`hpib2`	Second GPIB interface card.
`com1`	RS-232 interface card.

Examples: Opening an Interface Session

These examples open an interface session with an RS-232 interface.

C example:

```
INST com1;
com1 = iopen (“com1”);
```

Visual Basic example:

```
Dim com1 As Integer
com1 = iopen (“com1”)
```

Commander Sessions

A **commander session** allows your computer to talk to the interface controller. Typically, the controller is the computer used to communicate with devices on the interface. When the computer is not active controller, commander sessions can be used to talk to the computer that is active controller. In this mode, the computer is acting like a device on the interface.

Addressing Commander Sessions

To create a commander session, specify a valid interface address followed by a comma and then the string **`cmdr`** in the **`iopen`** function. The following are valid commander addresses:

`hpib,cmdr`	GPIB commander session.
`7,cmdr`	Commander session on interface at logical unit 7.

Examples: Creating a Commander Session

These examples create a commander session with the GPIB interface. The function calls open a session of communication with the commander on a GPIB interface.

C example:

```
INST cmdr;
cmdr = iopen(“hpib,cmdr”);
```

Visual Basic example:

```
Dim cmdr As Integer
cmdr = iopen (“hpib,cmdr”)
```

Sending I/O Commands

Once you have established a communications session with a device, interface, or commander, you can start communicating with that session using SICL's I/O routines. SICL provides formatted I/O and non-formatted I/O routines.

- **Formatted I/O** converts mixed types of data under the control of a format string. The data is buffered, thus optimizing interface traffic. The formatted I/O routines are geared towards instruments, and reduce the amount of I/O code.

- **Non-formatted I/O** sends or receives raw data to a device, interface, or commander. With non-formatted I/O, no format or conversion of the data is performed. Thus, if formatted data is required, it must be done by the user.

Formatted I/O in C Applications

The SICL formatted I/O mechanism is similar to the C **`stdio`** mechanism. SICL formatted I/O, however, is designed specifically for instrument communication and is optimized for IEEE 488.2 compatible instruments. The three main functions for formatted I/O in C applications are:

- The **`iprintf`** function formats according to the format string and sends data to a device:

  `iprintf(`*id, format [,arg1][,arg2][,...]*`);`

- The **`iscanf`** function receives and converts data according to the format string:

  `iscanf(`*id, format [,arg1][,arg2][,...]*`);`

- The **`ipromptf`** function formats and sends data to a device and then immediately receives and converts the response data:

  `ipromptf(`*id, writefmt, readfmt [,arg1][,arg2][,...]*`);`

The formatted I/O functions are buffered. Also, there are two non-buffered and non-formatted I/O functions called **`iread`** and **`iwrite`**. (See "Non-Formatted I/O" later in this chapter.) These are raw I/O functions and do not intermix with formatted I/O functions.

If raw I/O must be mixed, use the **ifread**/**ifwrite** functions. These functions have the same parameters as **iread** and **iwrite**, but read or write raw output data to the formatted I/O buffers. See “Formatted I/O Buffers” in this section for more details.

Formatted I/O Conversion

Formatted I/O functions convert data under the control of the format string. The format string specifies how the argument is converted before it is input or output. A typical format string syntax is:

```
%[format flags][field width][. precision][, array
size][argument modifier]conversion character
```

See **iprintf**, **ipromptf**, and **iscanf** in *Chapter 11 - SICL Language Reference* for more information on how data is converted under the control of the format string

Format Flags. Zero or more flags may be used to modify the meaning of the conversion character. The format flags are only used when sending formatted I/O (**iprintf** and **ipromptf**). Supported format flags are:

Format Flag	Description
@1	Converts to a 488.2 NR1 number.
@2	Converts to a 488.2 NR2 number.
@3	Converts to a 488.2 NR3 number.
@H	Converts to a 488.2 hexadecimal number.
@Q	Converts to a 488.2 octal number.
@B	Converts to a 488.2 binary number.
+	Prefixes number with sign (+ or –).
–	Left justifies result.
space	Prefixes number with blank space if positive or with – if negative.
#	Uses alternate form. For o conversion, it prints a leading zero. For x or X, a nonzero will have 0x or 0X as a prefix. For e, E, f, g, or G, the result will always have one digit on the right of the decimal point.
0	Causes left pad character to be a zero for all numeric conversion types.

This example converts **numb** into a 488.2 floating point number and sends the value to the session specified by *id*:

```
int numb = 61;
iprintf (id, "%@2d&\n", numb);
```

Sends: **61.000000**

Field Width. Field width is an optional integer that specifies how many characters are in the field. If the formatted data has fewer characters than specified in the field width, it will be padded. The padded character is dependent on various flags. You can use an asterisk (*****) in place of the integer to indicate that the integer is taken from the next argument.

This example pads **numb** to six characters and sends the value to the session specified by *id*:

```
long numb = 61;
iprintf (id, "%6ld&\n", numb);
```

Pads to six characters: **61**

. Precision. Precision is an optional integer preceded by a period. When used with conversion characters **e, E**, and **f**, the number of digits to the right of the decimal point are specified. For the **d, i, o, u, x,** and **X** conversion characters, the minimum number of digits to appear is specified. For the **s** and **S** conversion characters, the precision specifies the maximum number of characters to be read from the argument. This field is only used when sending formatted I/O (**iprintf** and **ipromptf**). You can use an asterisk (*****) in place of the integer to indicate that the integer is taken from the next argument.

This example converts **numb** so that there are only two digits to the right of the decimal point and sends the value to the session specified by *id*:

```
float numb = 26.9345;
iprintf (id, ".2f\n", numb);
```

Sends : **26.93**

, Array Size. The comma operator is a format modifier which allows you to read or write a comma-separated list of numbers (only valid with **%d** and **%f** conversion characters). It is a comma followed by an integer. The integer indicates the number of elements in the array. The comma operator has the format of **,** *dd* where *dd* is the number of elements to read or write.

This example specifies a comma-separated list to be sent to the session specified by *id*:

```
int list[5]={101,102,103,104,105};
iprintf (id, "%,5d\n", list);
```

Sends: **`101,102,103,104,105`**

Argument Modifier. The meaning of the optional argument modifier **`h, l, w, z,`** or **`Z`** is dependent on the conversion character.

Argument Modifiers in C Applications

Argument Modifier	Conversion Character	Description
`h`	**`d,i`**	Corresponding argument is a short integer.
`h`	**`f`**	Corresponding argument is a float for `iprintf` or a pointer to a float for `iscanf`.
`l`	**`d,i`**	Corresponding argument is a long integer.
`l`	**`b,B`**	Corresponding argument is a pointer to a block of long integers.
`l`	**`f`**	Corresponding argument is a double for `iprintf` or a pointer to a double for `iscanf`.
`w`	**`b,B`**	Corresponding argument is a pointer to a block of short integers.
`z`	**`b,B`**	Corresponding argument is a pointer to a block of floats.
`Z`	**`b,B`**	Corresponding argument is a pointer to a block of doubles.

Conversion Characters. The conversion characters for sending and receiving formatted I/O are different. The following tables summarize the conversion characters for each.

`iprintf` and `ipromptf` Conversion Characters in C Applications

Conversion Character	Description
`d,i`	Corresponding argument is an integer.
`f`	Corresponding argument is a float.
`b,B`	Corresponding argument is a pointer to an arbitrary block of data.
`c,C`	Corresponding argument is a character.
`t`	Controls whether the END indicator is sent with each LF character in the format string.
`s,S`	Corresponding argument is a pointer to a null terminated string.
`%`	Sends an ASCII percent (%) character.
`o,u,x,X`	Corresponding argument will be treated as an unsigned integer.
`e,E,g,G`	Corresponding argument is a double.
`n`	Corresponding argument is a pointer to an integer.
`F`	Corresponding argument is a pointer to a FILE descriptor opened for reading.

This example sends an arbitrary block of data to the session specified by the *id* parameter. The asterisk (**`*`**) is used to indicate that the number is taken from the next argument:

```
int size = 1024;
char data [1024];
.
.
iprintf (id, "%*b&\n", size, data);
```

Sends 1024 characters of block data.

`iscanf` and `ipromptf` Conversion Characters in C Applications

Conversion Character	Description
`d,i,n`	Corresponding argument must be a pointer to an integer.
`e,f,g`	Corresponding argument must be a pointer to a float.
`c`	Corresponding argument is a pointer to a character.
`s,S,t`	Corresponding argument is a pointer to a string.
`o,u,x`	Corresponding argument must be a pointer to an unsigned integer.
`[`	Corresponding argument must be a character pointer.
`F`	Corresponding argument is a pointer to a FILE descriptor opened for writing.

This example receives data from the session specified by the *id* parameter and converts the data to a string:

```
char data[180];
iscanf (id, "%s", data);
```

Example: Formatted I/O (C)

This C program example shows one way to send and receive formatted I/O. This example opens an GPIB communications session with a multimeter and uses a comma operator to send a comma-separated list to the multimeter. The *If* conversion characters are used to receive a double from the multimeter.

```
/* formatio.c
   This example program makes a multimeter measurement
   with a comma-separated list passed with formatted
   I/O and prints the results */
#include <sicl.h>
#include <stdio.h>
main()
{
   INST dvm;
   double res;
   double list[2] = {1,0.001};

#if defined(__BORLANDC__) && !defined(__WIN32__)
_InitEasyWin(); /*Required for Borland EasyWin programs*/
#endif
```

```
   /* Log message and terminate on error */
   ionerror (I_ERROR_EXIT);

   /* Open the multimeter session */
   dvm = iopen ("hpib7,16");
   itimeout (dvm, 10000);

   /*Initialize dvm*/
   iprintf (dvm, "*RST\n");

   /*Set up multimeter and send comma-separated list*/
   iprintf (dvm, "CALC:DBM:REF 50\n");
   iprintf (dvm, "MEAS:VOLT:AC? %,2lf\n", list);

   /* Read the results */
   iscanf (dvm,"%lf",&res);

   /* Print the results */
   printf ("Result is %f\n",res);

   /* Close the multimeter session */
   iclose (dvm);

/* This is a no-op for WIN32 programs.*/
   _siclcleanup();
  return 0;
}
```

Format String

The format string for **`iprintf`** puts a special meaning on the newline character (**`\n`**). The newline character in the format string flushes the output buffer to the device. All characters in the output buffer will be written to the device with an END indicator included with the last byte (the newline character). This means you can control at what point you want the data written to the device.

If no newline character is included in the format string for an **`iprintf`** call, the characters converted are stored in the output buffer. It will require another call to **`iprintf`** or a call to **`iflush`** to have those characters written to the device.

This can be very useful in queuing up data to send to a device. It can also raise I/O performance by doing a few large writes instead of several smaller

writes. This behavior can be changed by the **isetbuf** and **isetubuf** functions. See “Formatted I/O Buffers” for details.

The format string for **iscanf** ignores most white-space characters. Two white-space characters that it does not ignore are newlines (**\n**) and carriage returns (**\r**). These characters are treated just like normal characters in the format string, which *must* match the next non-white-space character read from the device.

Formatted I/O Buffers

The SICL software maintains both a read and write buffer for formatted I/O operations. Occasionally, you may want to control the actions of these buffers. See the **isetbuf** function for other options for buffering data.

The write buffer is maintained by the **iprintf** and the write portion of the **ipromptf** functions. It queues characters to send to the device so that they are sent in large blocks, thus increasing performance. The write buffer automatically flushes when it sends a newline character from the format string (see the **%t** conversion character to change this feature).

The write buffer also flushes immediately after the write portion of the **ipromptf** function. It may occasionally be flushed at other non-deterministic times, such as when the buffer fills. When the write buffer flushes, it sends its contents to the device.

The read buffer is maintained by the **iscanf** and the read portion of the **ipromptf** functions. The read buffer queues the data received from a device until it is needed by the format string. The read buffer is automatically flushed before the write portion of an **ipromptf**. Flushing the read buffer destroys the data in the buffer and guarantees that the next call to **iscanf** or **ipromptf** reads data directly from the device rather than data that was previously queued.

NOTE

Flushing the read buffer also includes reading all pending response data from a device. If the device is still sending data, the flush process will continue to read data from the device until it receives an **END** indicator from the device.

Related Formatted I/O Functions

A set of functions related to formatted I/O follows.

ifread	Obtains raw data directly from the read formatted I/O buffer. This is the same buffer that `iscanf` uses.
ifwrite	Writes raw data directly to the write formatted I/O buffer. This is the same buffer that `iprintf` uses.
iprintf	Converts data via a format string and writes the arguments appropriately.
iscanf	Reads data from a device/interface, converts this data via a format string, and assigns the values to your arguments.
ipromptf	Sends, then receives, data from a device/instrument. It also converts data via format strings that are identical to `iprintf` and `iscanf`.
iflush	Flushes the formatted I/O read and write buffers. A flush of the read buffer means that any data in the buffer is lost. A flush of the write buffer means that any data in the buffer is written to the session's target address.
isetbuf	Sets the size of the formatted I/O read and the write buffers. A size of zero (0) means no buffering. If no buffering is used, performance can be severely affected.
isetubuf	Sets the read or the write buffer to your allocated buffer. The same buffer cannot be used for both reading and writing. Also you should be careful when using buffers that are automatically allocated.

Formatted I/O in Visual Basic Applications

SICL formatted I/O is designed specifically for instrument communication and is optimized for IEEE 488.2 compatible instruments. The two main functions for formatted I/O in Visual Basic applications are:

- The **ivprintf** function formats according to the format string and sends data to a device:

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

- The **ivscanf** function receives and converts data according to the format string:

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

NOTE

There are certain restrictions when using **ivprintf** and **ivscanf** with Visual Basic. For details about these restrictions, see “Restrictions Using **ivprintf** in Visual Basic” in the **iprintf** function or “Restrictions Using **ivscanf** in Visual Basic” in the **iscanf** function in *Chapter 11 - SICL Language Reference*.

The formatted I/O functions are buffered. There are two non-buffered and non-formatted I/O functions called **iread** and **iwrite**. (See “Non-Formatted I/O” later in this chapter.) These are raw I/O functions and do not intermix with the formatted I/O functions.

If raw I/O must be mixed, use the **ifread**/**ifwrite** functions. They have the same parameters as **iread** and **iwrite**, but read or write raw output data to the formatted I/O buffers. See “Formatted I/O Buffers” for details.

Formatted I/O Conversion

The formatted I/O functions convert data under the control of the format string. The format string specifies how the argument is converted before it is input or output. The typical format string syntax is:

```
%[format flags][field width][. precision][, array
     size][argument modifier]conversion character
```

See **`iprintf`** and **`iscanf`** in *Chapter 11 - SICL Language Reference* for more information on how data is converted under the control of the format string.

Format Flags. Zero or more flags may be used to modify the meaning of the conversion character. The format flags are only used when sending formatted I/O (**`ivprintf`**). Supported format flags are:

Format Flags for `ivprintf` in Visual Basic Applications

Format Flag	Description
`@1`	Converts to a 488.2 NR1 number.
`@2`	Converts to a 488.2 NR2 number.
`@3`	Converts to a 488.2 NR3 number.
`@H`	Converts to a 488.2 hexadecimal number.
`@Q`	Converts to a 488.2 octal number.
`@B`	Converts to a 488.2 binary number.
`+`	Prefixes number with sign (+ or –).
`–`	Left justifies result.
`space`	Prefixes number with blank space if positive or with – if negative.
`#`	Uses alternate form. For o conversion, it prints a leading zero. For x or X, a nonzero will have 0x or 0X as a prefix. For e, E, f, g, or G, the result will always have one digit on the right of the decimal point.
`0`	Causes left pad character to be a zero for all numeric conversion types.

This example converts **numb** into a 488.2 floating point number to the session specified by *id*. The function return values must be assigned to variables for all Visual Basic function calls. Also, **+ Chr$(10)** adds the newline character to the format string to indicate that the formatted I/O write buffer should be flushed. (This is equivalent to the **\n** character sequence used for C/C++ programs.

```
Dim numb As Integer
Dim ret_val As Integer

numb = 61
ret_val = ivprintf(id, "%@2d" + Chr$(10), numb)
```

Sends: **61.000000**

Field Width. Field width is an optional integer that specifies how many characters are in the field. If the formatted data has fewer characters than specified in the field width, it will be padded. The padded character is dependent on various flags.

This example pads **numb** to six characters and sends the value to the session specified by *id*:

```
Dim numb As Integer
Dim ret_val As Integer

numb = 61
ret_val = ivprintf(id, "%6d" + Chr$(10), numb)
```

Pads to six characters: **61**

. Precision. Precision is an optional integer preceded by a period. When used with conversion characters **e, E**, and **f**, the number of digits to the right of the decimal point are specified. For the **d, i, o, u, x,** and **X** conversion characters, the minimum number of digits to appear is specified. This field is only used when sending formatted I/O (**ivprintf**).

This example converts *numb* so there are only two digits to the right of the decimal point and sends the value to the session specified by *id*:

```
Dim numb As Double
Dim ret_val As Integer
numb = 26.9345
ret_val = ivprintf(id, "%.2lf" + Chr$(10), numb)
```

Sends : **26.93**

, Array Size. The comma operator is a format modifier which allows you to read or write a comma-separated list of numbers (only valid with **%d** and **%f** conversion characters). It is a comma followed by an integer. The integer indicates the number of elements in the array. The comma operator has the format of **,** *dd* where *dd* is the number of elements to read or write.

This example specifies a comma separated list to be sent to the session specified by *id*:

```
Dim list(4) As Integer
Dim ret_val As Integer

list(0) = 101
list(1) = 102
list(2) = 103
list(3) = 104
list(4) = 105

ret_val = ivprintf(id, "%,5d" + Chr$(10), list(0))
```

Sends: **101,102,103,104,105**

Argument Modifier. The meaning of the optional argument modifier **h, l, w, z,** or **Z** is dependent on the conversion character.

Argument Modifiers in Visual Basic Application

Argument Modifier	Conversion Character	Description
h	**d,i**	Corresponding argument is an Integer.
h	**f**	Corresponding argument is a Single.
l	**d,i**	Corresponding argument is a Long.
l	**d,B**	Corresponding argument is an array of Long.
l	**f**	Corresponding argument is a Double.
w	**d,B**	Corresponding argument is an array of Integer.
z	**d,B**	Corresponding argument is an array of Single.
Z	**d,B**	Corresponding argument is an array of Double.

Conversion Characters. The conversion characters for sending and receiving formatted I/O are different. The following tables summarize the conversion characters for each:

`ivprintf` Conversion Characters in Visual Basic Applications

Conversion Character	Description
`d, i`	Corresponding argument is an Integer.
`b, B`	Not supported on Visual Basic.
`c,C`	Not supported on Visual Basic.
`t`	Not supported on Visual Basic.
`s,S`	Not supported on Visual Basic.
`%`	Sends an ASCII percent (%) character.
`o,u,x,X`	Corresponding argument will be treated as an Integer.
`f,e,E,g,G`	Corresponding argument is a Double.
`n`	Corresponding argument is an Integer.
`F`	Corresponding *arg* is a pointer to a FILE descriptor.

`ivscanf` Conversion Characters in Visual Basic Applications

Conversion Character	Description
`d,i,n`	Corresponding argument must be an Integer.
`e,f,g`	Corresponding argument must be a Single.
`c`	Corresponding argument is a fixed length String.
`s,S,t`	Corresponding argument is a fixed length String.
`o,u,x`	Corresponding argument must be an Integer.
`[`	Corresponding argument must be a fixed length character String.
`F`	Not supported on Visual Basic.

This example receives data from the session specified by the *id* parameter and converts the data to a string:

```
   Dim ret_val As Integer
   Dim data As String * 180

ret_val = ivscanf(id, "%180s", data)
```

Example: Formatted I/O (Visual Basic)

This Visual Basic program example sends and receives formatted I/O. The example opens a GPIB communications session with a multimeter and uses a comma operator to send a comma-separated list to the multimeter. The **`lf`** conversion characters are then used to receive a Double from the multimeter.

```
' formatio.bas
' The following subroutine makes a multimeter measurement with a comma-
' separated list passed with formatted I/O and prints the results.
'
Sub main()
   Dim dvm As Integer
   Dim res As Double
   ReDim list(2) As Double
   Dim nRetVal As Integer

   On Error GoTo ErrorHandler

   ' Initialize values in list
   list(0) = 1
   list(1) = 0.001

   ' Open the multimeter session
   dvm = iopen("hpib7,0")
   Call itimeout(dvm, 10000)

   ' Initialize dvm.
   nRetVal = ivprintf(dvm, "*RST" + Chr$(10), 0&)

   ' Set up multimeter and send comma separated list
   nRetVal = ivprintf(dvm, "CALC:DBM:REF 50" + Chr$(10))
   nRetVal = ivprintf(dvm, "MEAS:VOLT:AC? %,2lf" + Chr$(10), list())

   ' Read the results.
   nRetVal = ivscanf(dvm, "%lf", res)
```

```
    ‘ Display the results
    MsgBox “Result is “ + Format$(res)

    ‘ Close the multimeter session
    Call iclose(dvm)

    ‘ Tell SICL to clean up for this task
    Call siclcleanup
    End

ErrorHandler:
    ‘ Display the error message
    MsgBox “*** Error : “ + Error$, MB_ICON_EXCLAMATION
    ‘ Tell SICL to clean up for this task
    Call siclcleanup
    End
End Sub
```

Format String

In the format string for **`ivprintf`**, when the special characters **`Chr$(10)`** is used the output buffer to the device is flushed. All characters in the output buffer will be written to the device with an END indicator included with the last byte. This means you can control at what point you want the data written to the device.

If no **`Chr$(10)`** is included in the format string for an **`ivprintf`** call, the characters converted are stored in the output buffer. It will require another call to **`ivprintf`** or a call to **`iflush`** to have those characters written to the device. This can be very useful in queuing up data to send to a device. It can also raise I/O performance by doing a few large writes instead of several smaller writes.

The format string for **`ivscanf`** ignores most white-space characters. Two white-space characters that it does not ignore are newlines (**`Chr$(10)`**) and carriage returns (**`Chr$(13)`**). These characters are treated just like normal characters in the format string, which *must* match the next non-white-space character read from the device.

Formatted I/O Buffers

The SICL software maintains both a read and write buffer for formatted I/O operations. Occasionally, you may want to control the actions of these buffers.

The write buffer is maintained by the **`ivprintf`** function. It queues characters to send to the device so that they are sent in large blocks, thus increasing performance. The write buffer automatically flushes when it sends a newline character from the format string. The write buffer may occasionally be flushed at other non-deterministic times, such as when the buffer fills. When the write buffer flushes, it sends its contents to the device.

The read buffer is maintained by the **`ivscanf`** function. It queues the data received from a device until it is needed by the format string. Flushing the read buffer destroys the data in the buffer and guarantees that the next call to **`ivscanf`** reads data directly from the device rather than data that was previously queued.

> **NOTE**
>
> Flushing the read buffer also includes reading all pending response data from a device. If the device is still sending data, the flush process will continue to read data from the device until it receives an **`END`** indicator from the device.

Related Formatted I/O Functions

This set of functions are related to formatted I/O in Visual Basic:

`ifread`	Obtains raw data directly from the read formatted I/O buffer. This is the same buffer that `ivscanf` uses.
`ifwrite`	Writes raw data directly to the write formatted I/O buffer. This is the same buffer that `ivprintf` uses.
`ivprintf`	Converts data via a format string and converts the arguments appropriately.
`ivscanf`	Reads data from a device/interface, converts this data via a format string, and assigns the value to your arguments.
`iflush`	Flushes the formatted I/O read and write buffers. A flush of the read buffer means that any data in the buffer is lost. A flush of the write buffer means that any data in the buffer is written to the session's target address.

Non-Formatted I/O

There are two non-buffered, non-formatted I/O functions called **iread** and **iwrite**. These are raw I/O functions and do not intermix with the formatted I/O functions. If raw I/O must be mixed, use the **ifread** and **ifwrite** functions that have the same parameters as **iread** and **iwrite**, but read/write raw data from/to the formatted I/O buffers.

iread and **iwrite** Functions

The **iread** function reads raw data from the device or interface specified by the *id* parameter and stores the results in the location where *buf* is pointing.

C example:

```
iread(id, buf, bufsize, reason, actualcnt);
```

Visual Basic example:

```
Call iread(id, buf, bufsize, reason, actualcnt)
```

The **iwrite** function sends the data pointed to by *buf* to the interface or device specified by *id*:

C example:

```
iwrite(id, buf, datalen, end, actualcnt);
```

Visual Basic example:

```
Call iwrite(id, buf, datalen, end, actualcnt)
```

Example: Non-Formatted I/O (C)

This C language program illustrates using non-formatted I/O to communicate with a multimeter over the GPIB interface. The SICL non-formatted I/O functions **iwrite** and **iread** are used for communication. A similar example was used to illustrate formatted I/O earlier in this chapter.

```
/* nonfmt.c
   This example program measures AC voltage on a
   multimeter and prints the results*/

#include <sicl.h>
#include <stdio.h>

main()
{
   INST dvm;
   char strres[20];
   unsigned long actual;
```

```
  #if defined(__BORLANDC__) && !defined(__WIN32__)
  _InitEasyWin(); /*required for Borland EasyWin
programs*/
  #endif

  /* Log message and terminate on error */
  ionerror (I_ERROR_EXIT);

  /* Open the multimeter session */
  dvm = iopen ("hpib7,16");
  itimeout (dvm, 10000);

  /*Initialize dvm*/
  iwrite (dvm, "*RST\n", 5, 1, NULL);

  /*Set up multimeter and take measurements*/
  iwrite (dvm,"CALC:DBM:REF 50\n",16,1,NULL);
  iwrite (dvm,"MEAS:VOLT:AC? 1, 0.001\n",23,1,NULL);

  /* Read measurements */
  iread (dvm, strres, 20, NULL, &actual);

  /* NULL terminate result string and print the results*/
  /* This technique assumes the last byte sent was a line-
feed */
  if (actual){
     strres[actual - 1] = (char) 0;
     printf("Result is %s\n", strres);
  }
 /* Close the multimeter session */
  iclose(dvm);
  /* This call is a no-op for WIN32 programs.*/
  _siclcleanup();
return 0; }
```

Example: Non-Formatted I/O (Visual Basic)

```
‘  nonfmt.bas
‘  The following subroutine measures AC voltage on a
‘  multimeter and prints the results.
‘
Sub Main ()
  Dim dvm As Integer
  Dim strres As String * 20
  Dim actual As Long

‘  Open the multimeter session
  dvm = iopen(“hpib7,16”)
  Call itimeout(dvm, 10000)

‘  Initialize dvm
  Call iwrite(dvm,ByVal “*RST” + Chr$(10), 5, 1, 0&)

‘  Set up multimeter and take measurements
  Call iwrite(dvm,ByVal “CALC:DBM:REF 50” +
Chr$(10),16,1, 0&)

   Call iwrite(dvm,ByVal “MEAS:VOLT:AC? 1, 0.001” +
Chr$(10),23,1, 0&)

‘  Read measurements
  Call iread(dvm,ByVal strres, 20, 0&, actual)

‘  Print the results
  Print “Result is “ + Left$(strres, actual)

‘  Close the multimeter session
  Call iclose(dvm)

‘  Tell SICL to clean up for this task
  Call siclcleanup

   Exit Sub

End Sub
```

Handling Asynchronous Events

Asynchronous events are events that happen outside the control of your application. These events include Service ReQuests (**SRQ**s) and **interrupts**. An SRQ is a notification that a device requires service. Both devices and interfaces can generate SRQs and interrupts.

NOTE

SICL allows installation of SRQ and interrupt handlers in C programs, but does not support them in Visual Basic programs.

By default, asynchronous events are enabled. However, the library will not generate any events until the appropriate handlers are installed in your program.

If an application uses asynchronous events (**`ionsrq`**, **`ionintr`**), a callback thread is created by the underlying SICL implementation to service the asynchronous event. This thread will not be terminated until some other thread of the application performs an **`ExitProcess`** on Windows 95 or Windows 98 or calls **`iclose`** on Windows NT or Windows 2000.

Example declarations:

```
  void SICLCALLBACK my_int_handler(INST id, int reason,
long sec) {
    /* your code here */
  }

  void SICLCALLBACK my_srq_handler(INST id) {
    /* your code here */
  }
```

SRQ Handlers

The **ionsrq** function installs an SRQ handler. The currently installed SRQ handler is called any time its' corresponding device generates an SRQ. If an interface is unable to determine which device on the interface generated the SRQ, all SRQ handlers assigned to that interface will be called.

Therefore, an SRQ handler cannot assume that its' corresponding device generated an SRQ. The SRQ handler should use the **ireadstb** function to determine whether its device generated an SRQ. If two or more sessions refer to the same device, the handlers for each of the sessions are called.

Interrupt Handlers

Two distinct steps are required for an interrupt handler to be called. First, the interrupt handler must be installed. Second, the interrupt event or events need to be enabled. The **ionintr** function installs an interrupt handler. The **isetintr** function enables the interrupt event or events.

An interrupt handler can be installed with no events enabled. Conversely, interrupt events can be enabled with no interrupt handler installed. Only when both an interrupt handler is installed and interrupt events are enabled will the interrupt handler be called.

Temporarily Disabling/Enabling Asynchronous Events

To temporarily prevent *all* SRQ and interrupt handlers from executing, use the **iintroff** function to disable all asynchronous handlers for all sessions in the process.

To re-enable asynchronous SRQ and interrupt handlers previously disabled by **iintroff**, use the **iintron** function. This enables all asynchronous handlers for all sessions in the process that had been previously enabled. These functions do not affect the **isetintr** values or the handlers (**ionsrq** or **ionintr**). The default value for both functions is **on**.

For operating systems that support multiple threads such as Windows 95, Windows 98, Windows 2000, and Windows NT, SRQ and interrupt handlers execute on a separate thread (a thread created and managed by SICL). This means a handler can be executing when the **iintroff** call is made. If this occurs, the handler will continue to execute until it has completed.

An implication of this is that the SRQ or interrupt handler may need to synchronize its operation with the application's primary thread. This could be accomplished via WIN32 synchronization methods or by using SICL locks, where the handler uses a separate session to perform its work.

Calls to **iintroff**/**iintron** may be nested, meaning that there must be an equal number of ons and offs. Thus, calling the **iintron** function may not actually re-enable interrupts.

Occasionally, you may want to suspend a process and wait until an event occurs that causes a handler to execute. The **iwaithdlr** function causes the process to suspend until an enabled SRQ or interrupt condition occurs and the related handler executes. Once the handler completes its operation, this function returns and processing continues.

For this function to work properly, your application *must* turn interrupts off (i.e., use **iintroff**). The **iwaithdlr** function behaves as if interrupts are enabled. Interrupts are still disabled after the **iwaithdlr** function has completed.

Interrupts must be disabled if you use **iwaithdlr**. Use **iintroff** to disable interrupts. The reason for disabling interrupts is that there may be a race condition between the **isetintr** and **iwaithdlr.** If you only expect one interrupt, it might come before the **iwaithdlr**. This may or may not have the desired effect. For example:

```
...
ionintr (hpib, act_isr);
isetintr (hpib, I_INTR_INTFACT, 1);
...
iintroff ();
igpibpassctl (hpib, ba);
while (!done)
  iwaithdlr (0);
iintron ();
...
```

Handling Errors

This section gives guidelines to handle errors in SICL, including:

- Logging SICL Error Messages
- Using Error Handlers in C
- Using Error Handlers in Visual Basic

Logging SICL Error Messages

This section shows how to use the **Event Viewer** (Windows 2000 and Windows NT) or the **Message Viewer** (Windows 95 and Windows 98) to log SICL error messages.

- To use the **Event Viewer** (Windows 2000 and Windows NT), run the **Event Viewer** *after* you run the SICL program.
- To use the **Message Viewer** (Windows 95 and Windows 98), run the **Message Viewer** *before* you run the SICL program.

Using the Event Viewer

For Windows NT and Windows 2000, SICL logs internal messages as Windows NT/Windows 2000 events. This includes error messages logged by the **I_ERROR_EXIT** and **I_ERROR_NOEXIT** error handlers. While developing your SICL application or tracking down problems, you can view these messages by opening the the **Agilent IO Libraries Control** (on the taskbar) and clicking **Run Event Viewer**. Both system and application messages can be logged to the **Event Viewer** from SICL. SICL messages are identified by **SICL LOG** or by the driver name (e.g., **ag341i32**).

Using the Message Viewer

For Windows 95 or Windows 98, you can use the **Message Viewer** utility. This utility provides a debug window to which SICL logs internal messages during application execution, including those logged by the **I_ERROR_EXIT** and **I_ERROR_NOEXIT** error handlers. The **Message Viewer** utility provides menu selections for saving the logged messages to a file, and to clear the message buffer. To start the **Message Viewer** utility, open the **Agilent IO Libraries Control** (on the taskbar) and click **Run Message Viewer**.

Using Error Handlers in C

When a SICL function call in a C/C++ program results in an error, it typically returns a special value such as a NULL pointer or a non-zero error code. SICL allows you to install an error handler for all SICL functions within a C/C++ application to provide a convenient mechanism for handling errors.

Installing an error handler allows your application to ignore the return value, and permits the error procedure to detect errors and recover. The error handler is called before the function that generated the error completes. Eerror handlers are per process (*not* per session or per thread).

ionerror Function

The function **ionerror** used to install an error handler is defined as:

```
int ionerror (proc);
void (*proc)();
```

where:

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

The routine *proc* is the error handler and is called whenever a SICL error occurs. Two special reserved values of *proc* may be passed to the **ionerror** function

I_ERROR_EXIT	This value installs a special error handler which will log a diagnostic message and then terminate the process.
I_ERROR_NOEXIT	This value installs a special error handler which will log a diagnostic message and then allow the process to continue execution.

This mechanism has substantial advantages over other I/O libraries, because error handling code is located away from the center of your application.

Example: Installng an Error Handler (C)

Typically, error handling code is intermixed with the I/O code in an application. However, with SICL error handling routines no special error handling code is inserted between the I/O calls.

Instead, a single line at the top (calling **`ionerror`**) installs an error handler that gets called any time an error occurs. In this example, a standard, system-defined error handler is installed that logs a diagnostic message and then exits.

```
/* errhand.c
   This example demonstrates how a SICL error handler
   can be installed. */

#include <sicl.h>
#include <stdio.h>

main ()
{
   INST dvm;
   double res;

    #if defined(__BORLANDC__) && !defined(__WIN32__)
   _InitEasyWin(); /* Required for Borland EasyWin
programs */
   #endif

   ionerror (I_ERROR_EXIT);
   dvm = iopen ("hpib7,16");
   itimeout (dvm, 10000);
   iprintf (dvm, "%s\n", "MEAS:VOLT:DC?");
   iscanf (dvm, "%lf", &res);
   printf ("Result is %lf\n", res);
   iclose (dvm);

   /* This call is a no-op for WIN32 programs.*/
   _siclcleanup();

   return 0;
}
```

Example: Writing an Error Handler (C)

This is an example of writing and implementing your own error handler.

> **NOTE**
>
> If an error occurs in **`iopen`**, the *id* passed to the error handler may not be valid.

```
/* errhand2.c
   This program shows how you can install your own error
   handler*/
#include <sicl.h>
#include <stdio.h>
#include <stdlib.h>

void SICLCALLBACK err_handler (INST id, int error) {
   fprintf (stderr, "Error: %s\n", igeterrstr (error));
   exit (1);
}
main () {
   INST dvm;
   double res;

   #if defined(__BORLANDC__) && !defined(__WIN32__)
   _InitEasyWin(); /* Required for Borland EasyWin
programs */
   #endif

   ionerror (err_handler);
   dvm = iopen ("hpib7,16");
   itimeout (dvm, 10000);
   iprintf (dvm, "%s\n", "MEAS:VOLT:DC?");
   iscanf (dvm, "%lf", &res);
   printf ("Result is %lf\n", res);
   iclose (dvm);

   /* This call is a no-op for WIN32 programs*/
   _siclcleanup();

    return 0;
 }
```

Using Error Handlers in Visual Basic

Typically in an application, error handling code is intermixed with the I/O code. However, by using Visual Basic's error handling capabilities, no special error handling code need be inserted between the I/O calls. Instead, a single line at the top (**`On Error GoTo`**) installs an error handler in the subroutine that gets called any time a SICL or Visual Basic error occurs.

When a SICL call results in an error, the error is communicated to Visual Basic by setting Visual Basic's **`Err`** variable to the SICL error code and **`Error$`** is set to a human-readable string that corresponds to **`Err`**. This allows SICL to be integrated with Visual Basic's built-in error handling capabilities. SICL programs written in Visual Basic can set up error handlers with the Visual Basic **`On Error`** statement.

The SICL **`ionerror`** function for C programs is not used with Visual Basic. Similarly, the **`I_ERROR_EXIT`** and **`I_ERROR_NOEXIT`** default handlers used in C programs are not defined for Visual Basic.

When an error occurs within a Visual Basic program, the default behavior is to display a dialog box indicating the error and then halt the program. If you want your program to intercept errors and keep executing, you will need to install an error handler with the **`On Error`** statement. For example:

```
On Error GoTo MyErrorHandler
```

This will cause your program to jump to code at the label **`MyErrorHandler`** when an error occurs. Note that the error handling code must exist within the subroutine or function where the error handler was declared.

If you do not want to call an error handler or have your application terminate when an error occurs, you can use the **`On Error`** statement to tell Visual Basic to ignore errors. For example:

```
On Error Resume Next
```

This tells Visual Basic to proceed to the statement following the statement in which an error occurs. In this case, you could call the Visual Basic **`Err`** function in subsequent lines to find out which error occurred.

Visual Basic error handlers are only active within the scope of the subroutine or function in which they are declared. Each Visual Basic subroutine or function that wants an error handler must declare its own error handler. This is different than the way SICL error handlers installed with **`ionerror`** work in C programs. An error handler installed with **`ionerror`** remains active within the scope of the whole C program.

Example: Error Handlers (Visual Basic)

In this Visual Basic example, the error handler displays the error message in a dialog box and then terminates the program. When an error occurs, the Visual Basic **`Err`** variable is set to the error code and the **`Error$`** variable is set to the error message string for the error that occurred.

```
‘  errhand.bas
‘
Sub Main()
   Dim dvm As Integer
   Dim res As Double

   On Error GoTo ErrorHandler

   dvm = iopen(“hpib7,16”)
   Call itimeout(dvm, 10000)
   argcount = ivprintf(dvm, “MEAS:VOLT:DC?” + Chr$(10))
   argcount = ivscanf(dvm, “%lf”, res)
   MsgBox “Result is “ + Format(res)
   iclose (dvm)

    ‘ Tell SICL to clean up for this task
   Call siclcleanup
   End

ErrorHandler:
   ‘  Display the error message
     MsgBox “*** Error : “ + Error$, MB_ICON_EXCLAMATION
   ‘  Tell SICL to clean up for this task
   Call siclcleanup
   End
End Sub
```

Using Locks

Because SICL allows multiple sessions on the same device or interface, the action of opening does not mean you have exclusive use. In some cases this is not an issue, but should be a consideration if you are concerned with program portability.

What are Locks?

The SICL **`ilock`** function is used to **lock** an interface or device. The SICL **`iunlock`** function is used to unlock an interface or device.

Locks are performed on a per-session (device, interface, or commander) basis. Also, locks can be nested. The device or interface only becomes unlocked when the same number of unlocks are done as the number of locks. Doing an unlock without a lock returns the error **`I_ERR_NOLOCK`**.

What does it mean to lock? Locking an interface (from an interface session) restricts other device and interface sessions from accessing this interface. Locking a device restricts other device sessions from accessing this device; however, other interface sessions may continue to access the interface for this device. Locking a commander (from a commander session) restricts other commander sessions from accessing this commander.

CAUTION

It is possible for an interface session to access a device locked from a device session. In such a case, data may be lost from the device session that was underway. For example, *Agilent Visual Engineering Environment (VEE)* applications use SICL interface sessions. Therefore, I/O operations from VEE applications can supercede any device session that has a lock on a particular device.

Not all SICL routines are affected by locks. Some routines that set or return session parameters never touch the interface hardware and therefore work without locks. For information on using locks in multi-threaded SICL applications over LAN, see *Chapter 8 - Using SICL with LAN*.

Lock Actions

If a session tries to perform any SICL function that obeys locks on an interface or device currently locked by another session, the default action is to suspend the call until the lock is released or, if a timeout is set, until the call times out.

This action can be changed with the **`isetlockwait`** function (see *Chapter 11 - SICL Language Reference* for a description). If the **`isetlockwait`** function is called with the *flag* parameter set to 0 (zero), the default action is changed. Rather than causing SICL functions to suspend, an error will be returned immediately.

To return to the default action, to suspend and wait for an unlock, call the **`isetlockwait`** function with the *flag* set to any non-zero value.

Locking in a Multi-User Environment

In a multi-user/multi-process environment where devices are being shared, it is a good idea to use locking to ensure exclusive use of a particular device or set of devices. However, as explained in “Using Locks”, an interface session can access a device locked from a device session.

In general, it is not good programming practice to lock a device at the beginning of an application and unlock it at the end. This can result in deadlocks or long waits by others who want to use the resource.

The recommended procedure to use locking is per transaction. Per transaction means that you lock before you setup the device, then unlock after all desired data have been acquired. When sharing a device, you cannot assume the state of the device, so the beginning of each transaction should have any setup needed to configure the device or devices to be used.

Example: Locking (C Program)

```
/* locking.c
   This example shows how device locking can be
   used to gain exclusive access to a device*/

#include <sicl.h>
#include <stdio.h>

main()
{
   INST dvm;

   char strres[20];
   unsigned long actual;

   #if defined(__BORLANDC__) && !defined(__WIN32__)
   _InitEasyWin(); /* required for Borland EasyWin
                      programs */
   #endif

   /* Log message and terminate on error */
   ionerror (I_ERROR_EXIT);

   /* Open the multimeter session */
   dvm = iopen ("hpib7,16");
   itimeout (dvm, 10000);

   /* Lock the multimeter device to prevent access
      from other applications*/
   ilock(dvm);

   /* Take a measurement  */
   iwrite (dvm, "MEAS:VOLT:DC?\n", 14, 1, NULL);

   /* Read the results */
   iread (dvm, strres, 20, NULL, &actual);

   /* Release the multimeter device for use by others */
   iunlock(dvm);
   /* NULL terminate result string and print results */
   /* This technique assumes the last byte sent was a
      line-feed */
```

```
    if (actual) {
       strres[actual - 1] = (char) 0;
       printf("Result is %s\n", strres);
    }

    /* Close the multimeter session */
    iclose(dvm);

    /* This call is a no-op for WIN32 programs.*/
  _siclcleanup();

  return 0;
}
```

Example: Locking (Visual Basic)

```
' locking.bas
Sub Main ()
  Dim dvm As Integer
  Dim strres As String * 20
  Dim actual As Long

'  Install an error handler
  On Error GoTo ErrorHandler

'  Open the multimeter session
  dvm = iopen("hpib7,16")
  Call itimeout(dvm, 10000)

'  Lock the multimeter device to prevent access from other
applications
  Call ilock(dvm)

'  Take a measurement
  Call iwrite(dvm,ByVal "MEAS:VOLT:DC?" + Chr$(10), 14,
1, 0&)

'  Read the results
  Call iread(dvm,ByVal strres, 20, 0&, actual)

'  Release the multimeter device for use by others
  Call iunlock(dvm)
```

```
‘  Display the results
   MsgBox “Result is “ + Left$(strres, actual)

‘  Close the multimeter session
   Call iclose(dvm)

‘  Tell SICL to clean up for this task
   Call siclcleanup

   End

ErrorHandler:
   ‘  Display the error message.
      MsgBox “*** Error : “ + Error$
   ‘  Tell SICL to clean up for this task
      Call siclcleanup

   End

End Sub
```

4

Using SICL with GPIB

Using SICL with GPIB

This chapter shows how to open a communications session and communicate with GPIB devices, interfaces, or controllers. The example programs in this chapter are also provided in the **C\SAMPLES\MISC** (for C/C++) or **VB\SAMPLES\MISC** (for Visual Basic) of the **IO Libraries** base directory. This chapter includes:

- Introduction
- Using GPIB Device Sessions
- Using GPIB Interface Sessions
- Using GPIB Commander Sessions
- Writing GPIB Interrupt Handlers

Introduction

This section provides an introduction to using SICL with the GPIB interface, including:

- Selecting a GPIB Communications Session
- SICL GPIB Functions

Selecting a GPIB Communications Session

When you have determined the GPIB system is set up and operating correctly, you can start programming with the SICL functions. First, you must determine what type of communications session to use.

The three types of communications sessions are **device**, **interface**, and **commander**. To use a device session, see “Using GPIB Device Sessions”. To use an interface session, see “Using GPIB Interface Sessions”. To use a commander session, see “Using GPIB Commander Sessions”.

SICL GPIB Functions

Function Name	Action
igpibatnctl	Sets or clears the ATN line
igpibbusaddr	Changes bus address
igpibbusstatus	Returns requested bus data
igpibgett1delay	Returns the current T1 setting for the interface
igpibllo	Sets bus in Local Lockout Mode
igpibpassctl	Passes active control to specified address
igpibppoll	Performs a parallel poll on the bus
igpibppollconfig	Configures device for PPOLL response
igpibppollresp	Sets PPOLL state
igpibrenctl	Sets or clears the REN line
igpibsendcmd	Sends data with ATN line set
igpibsett1delay	Sets the T1 delay value for this interface

Using GPIB Device Sessions

A **device session** allows you direct access to a device without knowing the type of interface to which it is connected. The specifics of the interface are hidden from the user.

Addressing GPIB Devices

To create a device session, specify the interface logical unit or symbolic name and a particular device logical address in the *addr* parameter of the **`iopen`** function. The interface logical unit and symbolic name are set by running the **`IO Config`** utility. To open **`IO Config,`** open the **`Agilent IO Libraries Control`** (on the taskbar) and click **`Run IO Config`**. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on the **`IO Config`** utility.

Primary and Secondary Addresses

SICL supports both primary and secondary addressing on GPIB interfaces. The primary address must be between 0 and 30 and the secondary address must be between 0 and 30. The primary and secondary addresses correspond to the GPIB primary and secondary addresses. Some example GPIB addresses for device sessions are:

`GPIB,7`	A device address corresponding to the device at primary address 7
`hpib,3,2`	A device address corresponding to the device at primary address 3, secondary address 2

VXI Mainframe Connections

For connections to a VXI mainframe via an E1406 Command Module (or equivalent), the primary address passed to **`iopen`** corresponds to the address of the Command Module and the secondary address must be specified to select a specific instrument in the card cage.

Secondary addresses of 0, 1, 2, ... 30 correspond to VXI instruments at logical addresses of 0, 8, 16, ... 240, respectively. See “GPIB Device Session Examples” in this chapter for an example program to communicate with a VXI mainframe via the GPIB interface.

Examples to open a device session with an GPIB device at bus address 16 follow.

C example:

```
INST dmm;
dmm = iopen (“hpib,16”);
```

Visual Basic example:

```
Dim dmm As Integer
dmm = iopen (“hpib,16”)
```

SICL Function Support for GPIB Device Session

This section shows how some SICL functions are implemented for GPIB device sessions. The data transfer functions work only when the GPIB interface is the Active Controller. Passing control to another GPIB device causes this device to lose active control.

iwrite	Causes all devices to untalk and unlisten. It sends this controller’s talk address followed by unlisten and then the listen address of the corresponding device session. Then, it sends the data over the bus.
iread	Causes all devices to untalk and unlisten. It sends an unlisten, then sends this controller’s listen address followed by the talk address of the corresponding device session. Then, it reads the data from the bus.
ireadstb	Performs a GPIB serial poll (SPOLL).
itrigger	Performs an addressed GPIB group execute trigger (GET).
iclear	Performs a GPIB selected device clear (SDC) on the device corresponding to this session.

GPIB Device Sessions and Service Requests

There are no device-specific interrupts for the GPIB interface, but GPIB device sessions do support Service Requests (SRQs). On the GPIB interface, when one device issues an SRQ, the library informs *all* GPIB device sessions that have SRQ handlers installed (see **`ionsrq`** in *Chapter 11 - SICL Language Reference*).

This is an artifact of how GPIB handles the SRQ line. The interface cannot distinguish which device requested service. Therefore, the library acts as if all devices require service. The SRQ handler can retrieve the device's **status byte** by using the **`ireadstb`** function. For more information, see "Writing GPIB Interrupt Handlers" in this chapter.

Example: GPIB Device Session (C)

This example opens two GPIB communications sessions with VXI devices (via a VXI Command Module). Then, a scan list is sent to a switch and measurements are taken by the multimeter every time a switch is closed.

```
/* hpibdev.c
   This example program sends a scan list to a switch
   and, while looping, closes channels and takes
   measurements. */

#include <sicl.h>
#include <stdio.h>

main()
{
   INST dvm;
   INST sw;

double res;
int i;

#if defined(__BORLANDC__) && !defined(__WIN32__)
_InitEasyWin(); /* Required for Borland EasyWin
                   programs */
#endif

/* Log message and terminate on error */
ionerror (I_ERROR_EXIT);

/* Open the multimeter and switch sessions*/
dvm = iopen ("hpib7,9,3");
```

```
sw = iopen (“hpib7,9,14”);
itimeout (dvm, 10000);
itimeout (sw, 10000);

/*Set up trigger*/
iprintf (sw, “TRIG:SOUR BUS\n”);

/*Set up scan list*/
iprintf (sw,”SCAN (@100:103)\n”);
iprintf (sw,”INIT\n”);

for (i=1;i<=4;i++)
{
   /* Take a measurement */
   iprintf (dvm,”MEAS:VOLT:DC?\n”);

   /* Read the results */
   iscanf (dvm,”%lf”,&res);

   /* Print the results */
   printf (“Result is %lf\n”,res);

   /* Trigger to close channel */
   iprintf (sw, “TRIG\n”);
}
/* Close the multimeter and switch sessions */
iclose (dvm);
iclose (sw);

/* This call is a no-op for WIN32 programs*/
_siclcleanup();

   return 0;
}
```

Example: GPIB Device Session (Visual Basic)

This example opens two GPIB communications sessions with VXI devices (via a VXI Command Module). Then, a scan list is sent to a switch and measurements are taken by the multimeter every time a switch is closed.

```
'hpibdev.bas
' This example program sends a scan list to a switch and
' while looping closes channels and takes measurements.

Sub Main ()
   Dim dvm As Integer
   Dim sw As Integer
   Dim res As Double
   Dim i As Integer
   Dim argcount As Integer

   ' Open the multimeter and switch sessions
   dvm = iopen("hpib7,9,3")
   sw = iopen("hpib7,9,14")
   Call itimeout(dvm, 10000)
   Call itimeout(sw, 10000)

   ' Set up trigger
   argcount = ivprintf(sw, "TRIG:SOUR BUS" + Chr$(10))

   ' Set up scan list
   argcount = ivprintf(sw, "SCAN (@100:103)" + Chr$(10))

   argcount = ivprintf(sw, "INIT" + Chr$(10))

   ' Display form1 and print voltage measurements
   form1.Show

   For i = 1 To 4
      ' Take a measurement
      argcount = ivprintf(dvm, "MEAS:VOLT:DC?" + Chr$(10))

      ' Read the results
      argcount = ivscanf(dvm, "%lf", res)

      ' Print the results
      form1.Print "Result is " + Format(res)
```

```
      ‘ Trigger to close channel
      argcount = ivprintf(sw, “TRIG” + Chr$(10))
   Next i

‘ Close the voltmeter session
Call iclose(dvm)

‘ Close the switch session
Call iclose(sw)

‘ Tell SICL to clean up for this task
Call siclcleanup

End Sub
```

Using GPIB Interface Sessions

Interface sessions allow direct, low-level control of the specified interface, but the programmer must provide all bus maintenance settings for the interface and must know the technical details about the interface. Also, when using interface sessions, interface-specific functions must be used. Thus, the program cannot be used on other interfaces and becomes less portable.

Addressing GPIB Interfaces

To create an interface session on your GPIB system, specify the particular interface logical unit or symbolic name in the *addr* parameter of the **iopen** function. The interface logical unit and symbolic name are set by running the **IO Config** utility. To open **IO Config,** open the **Agilent IO Libraries Control** (on the taskbar) and click **Run IO Config**. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on the **IO Config** utility. Example interface addresses follow.

GPIB	An interface symbolic name.
hpib	An interface symbolic name.
gpib2	An interface symbolic name.
IEEE488	An interface symbolic name.
7	An interface logical unit.

These examples open an interface session with the GPIB interface.

C example:

```
INST hpib;
hpib = iopen (“hpib”);
```

Visual Basic example:

```
Dim hpib As Integer
hpib = iopen (“hpib”)
```

SICL Function Support for GPIB Interface Sessions

This section describes how some SICL functions are implemented for GPIB interface sessions.

`iwrite`	Sends the specified bytes directly to the interface without performing any bus addressing. The `iwrite` function always clears the ATN line before sending any bytes, thus ensuring that the GPIB interface sends the bytes as data, not as command bytes.
`iread`	Reads the data directly from the interface without performing any bus addressing.
`itrigger`	Performs a broadcast GPIB group execute trigger (GET) without additional addressing. Use this function with `igpibsendcmd` to send a `UNL` followed by the appropriate device addresses. This will allow the `itrigger` function to be used to trigger multiple GPIB devices simultaneously. Passing the `I_TRIG_STD` value to the ixtrig function also causes a broadcast GPIB group execute trigger (GET). There are no other valid values for the `ixtrig` function.
`iclear`	Performs a GPIB interface clear (pulses IFC), which resets the interface.

GPIB Interface Sessions Interrupts

There are specific interface session interrupts that can be used. See **`isetintr`** in *Chapter 11 - SICL Language Reference* for information on the interface session interrupts for GPIB. Also, see “Writing GPIB Interrupt Handlers” in this chapter for more information.

GPIB Interface Sessions and Service Requests

GPIB interface sessions support Service Requests (SRQs). On the GPIB interface, when one device issues an SRQ, the library will inform *all* GPIB interface sessions that have SRQ handlers installed (see **`ionsrq`** in *Chapter 11 - SICL Language Reference*). For more information, see “Writing GPIB Interrupt Handlers” in this chapter.

Example: GPIB Interface Session (C)

```
/* hpibstat.c
   This example retrieves and displays GPIB
   bus status information.  */

#include <stdio.h>
#include <sicl.h>

main()
{
   INST id;       /* session id        */
   int rem;       /* remote enable     */
   int srq;       /* service request   */
   int ndac;      /* not data accepted */
   int sysctlr;   /* system controller */
   int actctlr;   /* active controller */
   int talker;    /* talker            */
   int listener;  /* listener          */
   int addr;      /* bus address       */

   #if defined(__BORLANDC__) && !defined(__WIN32__)
       _InitEasyWin(); /* Required for Borland EasyWin programs */
   #endif

   /* exit process if SICL error detected */
   ionerror(I_ERROR_EXIT);

   /* open GPIB interface session */
   id = iopen("hpib");
itimeout (id, 10000);

   /* retrieve GPIB bus status */
   igpibbusstatus(id, I_GPIB_BUS_REM,      &rem);
   igpibbusstatus(id, I_GPIB_BUS_SRQ,      &srq);
   igpibbusstatus(id, I_GPIB_BUS_NDAC,     &ndac);
   igpibbusstatus(id, I_GPIB_BUS_SYSCTLR,  &sysctlr);
   igpibbusstatus(id, I_GPIB_BUS_ACTCTLR,  &actctlr);
   igpibbusstatus(id, I_GPIB_BUS_TALKER,   &talker);
   igpibbusstatus(id, I_GPIB_BUS_LISTENER, &listener);
   igpibbusstatus(id, I_GPIB_BUS_ADDR,     &addr);
```

```
    /* display bus status */
    printf("%-5s%-5s%-5s%-5s%-5s%-5s%-5s%-5s\n",
    "REM", "SRQ","NDC", "SYS", "ACT", "TLK", "LTN",
    "ADDR");printf("%2d%5d%5d%5d%5d%5d%5d%6d\n",
    rem, srq, ndac, sysctlr, actctlr, talker, listener,
    addr);

   /* This call is no-op for WIN32 programs.*/

  _siclcleanup();

  return 0;
}
```

Example: GPIB Interface Session (Visual Basic)

```
  'hpibstat.bas
  '  The following example retrieves and displays
  '  GPIB bus status information.
  Sub main ()
    Dim id As Integer   ' session id
    Dim remen As Integer  ' remote enable
    Dim srq As Integer    ' service request
    Dim ndac As Integer   ' not data accepted
    Dim sysctlr As Integer' system controller
    Dim actctlr As Integer' active controller
    Dim talker As Integer ' talker
    Dim listener As Integer' listener
    Dim addr As Integer   ' bus address
    Dim header As String  ' report header
    Dim values As String  ' report output

  ' Open GPIB interface session
    id = iopen("hpib7")
    Call itimeout(id, 10000)

  ' Retrieve GPIB bus status
    Call igpibbusstatus(id, I_GPIB_BUS_REM, remen)
    Call igpibbusstatus(id, I_GPIB_BUS_SRQ, srq)
    Call igpibbusstatus(id, I_GPIB_BUS_NDAC, ndac)
    Call igpibbusstatus(id, I_GPIB_BUS_SYSCTLR, sysctlr)
    Call igpibbusstatus(id, I_GPIB_BUS_ACTCTLR, actctlr)
    Call igpibbusstatus(id, I_GPIB_BUS_TALKER, talker)
    Call igpibbusstatus(id, I_GPIB_BUS_LISTENER, listener)
```

```
    Call igpibbusstatus(id, I_GPIB_BUS_ADDR, addr)

  ‘ Display form1 and print results
    form1.Show
    form1.Print “REM”; Tab(7); “SRQ”; Tab(14); “NDC”;
    Tab(21);“SYS”; Tab(28); “ACT”; Tab(35); “TLK”;
    Tab(42); “LTN”; Tab(49);“ADDR” form1.Print remen;
    Tab(7); srq; Tab(14); ndac; Tab(21);sysctlr;
    Tab(28); actctlr; Tab(35); talker; Tab(42);
    listener; Tab(49); addr

    ‘ Tell SICL to clean up for this task
    Call siclcleanup

End Sub
```

Using GPIB Commander Sessions

Commander sessions are intended for use on GPIB interfaces that are not the active controller. In this mode, a computer that is not the controller is acting like a device on the GPIB bus. In a commander session, the data transfer routines only work when the GPIB interface is not active controller.

Addressing GPIB Commanders

To create a commander session on your GPIB interface, specify the particular interface logical unit or symbolic name in the *addr* parameter followed by a comma and the string **`cmdr`** in the **`iopen`** function.

The interface logical unit and symbolic name are set by running the **`IO Config`** utility. To open **`IO Config,`** open the **`Agilent IO Libraries Control`** (on the taskbar) and click **`Run IO Config`**. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on the **`IO Config`** utility. Example GPIB addresses for commander sessions follow.

`GPIB,cmdr`	A commander session with the GPIB interface.
`hpib2,cmdr`	A commander session with the hpib2 interface.
`7,cmdr`	A commander session with the interface at logical unit 7.

These examples open a commander session with the GPIB interface.

C example:

```
INST hpib;
hpib = iopen (“hpib,cmdr”);
```

Visual Basic example:

```
Dim hpib As Integer
hpib = iopen (“hpib,cmdr”)
```

SICL Function Support for GPIB Commander Sessions

This section describes how some SICL functions are implemented for GPIB commander sessions.

iwrite	If the interface has been addressed to talk, the data is written directly to the interface. If the interface has not been addressed to talk, it will wait to be addressed to talk before writing the data.
iread	If the interface has been addressed to listen, the data is read directly from the interface. If the interface has not been addressed to listen, it will wait to be addressed to listen before reading the data.
isetstb	Sets the status value that will be returned on a `ireadstb` call (that is, when this device is SPOLLed). Bit 6 of the status byte has a special meaning. If bit 6 is set, the SRQ line will be set. If bit 6 is clear, the SRQ line will be cleared.

GPIB Commander Sessions Interrupts

There are specific commander session interrupts that can be used. See **isetintr** in *Chapter 11 - SICL Language Reference* for information on commander session interrupts. Also see “Writing GPIB Interrupt Handlers” for more information.

Writing GPIB Interrupt Handlers

This section provides some additional information for writing interrupt handlers for GPIB applications in SICL.

Multiple `I_INTR_GPIB_TLAC` Interrupts

This interrupt occurs whenever a device has been addressed to talk or untalk, or a device has been addressed to listen or unlisten. Due to hardware limitations, your SICL interrupt handler may be called twice in response to any of these events.

Your GPIB application should be written to handle this situation gracefully. This can be done by keeping track of the current talk/listen state of the interface card and ignoring the interrupt if the state does not change. For more information, see the *secval* parameter definition of the **`isetintr`** function in *Chapter 11 - SICL Language Reference*.

Handling SRQs from Multiple GPIB Instruments

GPIB is a multiple-device bus and SICL allows multiple device sessions open at the same time. On the GPIB interface, when one device issues a Service Request (SRQ), the library will inform *all* GPIB device sessions that have SRQ handlers installed (see **`ionsrq`** in *Chapter 11 - SICL Language Reference*).

This is an artifact of how GPIB handles the SRQ line. The underlying GPIB hardware does not support session-specific interrupts like VXI does. Therefore, your application must reflect the nature of the GPIB hardware if you expect to reliably service SRQs from multiple devices on the same GPIB interface.

It is vital that you never exit an SRQ handler without first clearing the SRQ line. If the multiple devices are all controlled by the same process, the easiest technique is to service all devices from one handler. The pseudo-code for this follows. This algorithm loops through all the device sessions and does not exit until the SRQ line is released (not asserted).

```
while (srq_asserted) {
   serial_poll (device1)
   if (needs_service) service_device1
   serial_poll (device2)
   if (needs_service) service_device2
```

```
    ...
    check_SRQ_line
  }
```

Example: Servicing Requests (C)

This example shows a SICL program segment that implements this algorithm. Checking the state of the SRQ line requires an interface session. Only one device session needs to execute **`ionsrq`** because that handler is invoked regardless of which instrument asserted the SRQ line. Assuming IEEE-488 compliance, an **`ireadstb`** is all that is needed to clear the device's SRQ.

Since the program cannot leave the handler until all devices have released SRQ, it is recommended that the handler do as little as possible for each device. The previous example assumed that only one **`iscanf`** was needed to service the SRQ. If lengthy operations are needed, a better technique is to perform the **`ireadstb`** and set a flag in the handler. Then, the main program can test the flags for each device and perform the more lengthy service.

Even if the different device sessions are in different processes, it is still important to stay in the SRQ handler until the SRQ line is released. However, it is not likely that a process which only knows about Device A can do anything to make Device B release the SRQ line.

In such a configuration, a single unserviced instrument can effectively disable SRQs for all processes attempting to use that interface. Again, this is a hardware characteristic of GPIB. The only way to ensure true independence of multiple GPIB processes is to use multiple GPIB interfaces.

```
/* Must be global */
INST id1, id2, bus;

void handler (dummy)
INST dummy;
{
  int srq_asserted = 1;
  unsigned char statusbyte;

/* Service all sessions in turn until no one is
   requesting service */
while (srq_asserted) {
  ireadstb(id1, &statusbyte);
  if (statusbyte & SRQ_BIT)
  {
```

```
      /* Actual service actions depend upon application */
      iscanf(id1, "%f", &data1);
   }
   ireadstb(id2, &statusbyte);
   if (statusbyte & SRQ_BIT){
   iscanf(id2, "%f", &data2);
   }
   igpibbusstatus(bus, I_GPIB_BUS_SRQ, &srq_asserted);
   }
}

main() {
   .
   .
   /* Device sessions for instruments */
   id1 = iopen("hpib, 17");
   id2 = iopen("hpib, 18");

   /* Interface session for SRQ test */
   bus = iopen("hpib");

   /* Only one handler needs to be installed */
   ionsrq(id1, handler);
   .
   .
```

Notes:

5

Using SICL with GPIO

Using SICL with GPIO

This chapter shows how to open an interface communications session and communicate with an instrument over a GPIO connection. The example programs in this chapter are also provided in the **C\SAMPLES\MISC** (for C/C++) and **VB\SAMPLES\MISC** (for Visual Basic) subdirectories. This chapter includes:

- Introduction
- Using GPIO Interface Sessions

Introduction

This section provides an introduction to using SICL with the GPIO interface, including:

- Selecting a GPIO Communications Session
- SICL GPIO Functions

Selecting a GPIO Communications Session

GPIO is a parallel interface that is flexible and allows a variety of custom connections. Although GPIO typically requires more time to configure than GP-IB, the speed and versatility of GPIO make it the perfect choice for many tasks.

NOTE

GPIO is *only* supported with SICL on Windows 95, Windows 98, Windows 2000, and Windows NT. GPIO is *not* supported with SICL via LAN.

Once you have configured your system for GPIO communications, you can start programming with the SICL functions. If you have programmed GPIO before, you will probably want to open the interface and start sending commands.

With GPIB, there can be multiple devices on a single interface. These interfaces support a connection called a **device session**. With GPIO, only one device is connected to the interface. Therefore, communication with GPIO devices must be using an **interface session**.

SICL GPIO Functions

Function Name	Action
`igpioctrl`	Sets the following characteristics of the GPIO interface:

Request	Characteristic	Settings
I_GPIO_AUTO_HDSK	Auto-Handshake mode	1 or 0
I_GPIO_AUX	Auxiliary Control lines	16-bit mask
I_GPIO_CHK_PSTS	Check PSTS before read/write	1 or 0
I_GPIO_CTRL	Control lines	**I_GPIO_CTRL_CTL0** **I_GPIO_CTRL_CTL1**
I_GPIO_DATA	Data Output lines	8-bit or 16-bit mask
I_GPIO_PCTL_DELAY	PCTL delay time	0-7
I_GPIO_POLARITY	Logical polarity	0-31
I_GPIO_READ_CLK	Data input latching	See *Chapter 11 - SICL Language Reference*
I_GPIO_READ_EOI	END termination pattern	**I_GPIO_EOI_NONE** or 8-bit or 16-bit mask
I_GPIO_SET_PCTL	Start PCTL handshake	1

igpiogetwidth	Returns the current width (in bits) of the GPIO data ports.
igpiosetwidth	Sets the width (in bits) of the GPIO data ports. Either 8 or 16.

igpiostat	Gets the following information about the GPIO interface:

Request	Characteristic	Value
I_GPIO_CTRL	Control Lines	**I_GPIO_CTRL_CTL0** **I_GPIO_CTRL_CTL1**
I_GPIO_DATA	Data In lines	16-bit mask
I_GPIO_INFO	GPIO information	**I_GPIO_AUTO_HDSK** **I_GPIO_CHK_PSTS** **I_GPIO_EIR** **I_GPIO_ENH_MODE** **I_GPIO_PSTS** **I_GPIO_READY**
I_GPIO_READ_EOI	END termination pattern	**I_GPIO_EOI_NONE** or 8-bit or 16-bit mask
I_GPIO_STAT	Status lines	**I_GPIO_STAT_STI0** **I_GPIO_STAT_STI1**

Using GPIO Interface Sessions

GPIO Interface sessions are used for GPIO data transfer, interrupt, status, and control operations. When communicating with a GPIO interface session, the programmer must specify the interface name.

Addressing GPIO Interfaces

To create an interface session on GPIO, specify the interface logical unit or symbolic name in the *addr* parameter of the **`iopen`** function. The interface logical unit and symbolic name are defined by running the **`IO Config`** utility. To open **`IO Config`**, click the **`Agilent IO Libraries Control`** (on the taskbar) and click **`Run IO Config`**. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on **`IO Config`**. Some example addresses for GPIO interface sessions are:

`gpio`	An interface symbolic name
`12`	An interface logical unit

This example opens an interface session with the GPIO interface.

```
INST intf;
intf = iopen (“gpio”);
```

SICL Function Support with GPIO Interface Sessions

This section describes how some SICL functions are implemented for GPIO interface sessions.

GPIO Interface Sessions SICL Functions

iwrite, iread	The *size* parameters for non-formatted I/O functions are always byte counts, regardless of the current data width of the interface.
iprintf, iscanf	All formatted I/O functions work with GPIO. When formatted I/O is used with 16-bit data widths, the formatting buffers reassemble the data as a stream of bytes. On Windows, these bytes are ordered: high-low-high-low... Because of this "unpacking" operation, 16-bit data widths may not be appropriate for formatted I/O operations. For **iscanf** termination, an END value must be specified using **igpioctrl.** See *Chapter 11 - SICL Language Reference*.
itermchr	For 16-bit data widths, only low (least-significant) byte is used.
ixtrig	Provides a method of triggering using either the CTL0 or CTL1 control lines. This function pulses the specified control line for approximately 1 or 2 µsec. The following constants are defined: **I_TRIG_STD** Pulse CTL0 line **I_TRIG_GPIO_CTL0** Pulse CTL0 line **I_TRIG_GPIO_CTL1** Pulse CTL1 line
itrigger	Same as **ixtrig** (I_TRIG_STD). Pulses the CTL0 control line.
iclear	Pulses the P_RESET line for at least 12 µsec, aborts any pending writes, discards any data in the receive buffer, and resets any error conditions. Optionally, clears the Data Out port, depending on the configuration specified via **IO Config**.
ionsrq	Installs a service request handler for this session. The concept of service request (SRQ) originates from GPIB. On a GPIB interface, a device can request service from the controller by asserting a line on the interface bus. On GPIO, the EIR line is assumed to be the service request line.
ireadstb	Although **ireadstb** is for device sessions only, since GPIO has no device sessions, **ireadstb** is allowed with GPIO interface sessions. The interface status byte has bit 6 set if EIR is asserted. Otherwise, the status byte is 0 (zero). This allows normal SRQ programming techniques in GPIO SRQ handlers.

GPIO Interface Sessions Interrupts

There are specific interface session interrupts that can be used. See **isetintr** in *Chapter 11 - SICL Language Reference* for information on the interface session interrupts for GPIO.

Example: GPIO Interface Session (C)

```
/* gpiomeas.c
This program:
- Creates a GPIO session with timeout and error checking
- Signals the device with a CTL0 pulse
- Reads the device's response using formatted I/O */

#include <sicl.h>

main()
{
   INST id;       /* interface session id */
   float result;  /* data from device */

#if defined (__BORLANDC__) && !defined (__WIN32__)
_InitEasyWin(); /* required for Borland EasyWin programs */
#endif

/* log message and exit program on error */
ionerror(I_ERROR_EXIT);

/* open GPIO interface session, with 3 sec timeout*/
id = iopen("gpio");
itimeout(id, 3000);

/* setup formatted I/O configuration */
igpiosetwidth(id, 8);
igpioctrl(id, I_GPIO_READ_EOI, '\n');

/* monitor the device's PSTS line */
igpioctrl(id, I_GPIO_CHK_PSTS, 1);

/* signal the device to take a measurement */
itrigger(id);

/* get the data */
iscanf(id, "%f%*t", &result);
printf("Result = %f\n", result);
/* This call is a no-op for WIN32 applications.*/
_siclcleanup();

/* close session */
iclose (id); }
```

Example: GPIO Interface Session (Visual Basic)

```
‘ This program:
‘ - Creates a GPIO session with timeout and error checking
‘ - Signals the device with a CTL0 pulse
‘ - Reads the device’s response using formatted I/O
‘
Sub cmdMeas_Click ()
   Dim id As Integer              ‘ device session id
   Dim retval As Integer          ‘ function return value
   Dim buf As String              ‘ buffer for displaying
   Dim real_data As Double        ‘ data from device

‘  Set up an error handler within this subroutine that will
‘  be called if a SICL error occurs.
   On Error GoTo ErrorHandler

‘  Disable the button used to initiate I/O while I/O is
‘  being performed.
   cmdMeas.Enabled = False

‘  Open an interface session using a known symbolic name
   id = iopen(“gpio12”)

‘  Set the I/O timeout value for this session to 3 sec
   Call itimeout(id, 3000)

‘  Setup formatted I/O configuration
   Call igpiosetwidth(id, 8)
   Call igpioctrl(id, I_GPIO_READ_EOI, 10)

‘  Signal the device to take a measurement
   Call itrigger(id)

‘  Get the data
   retval = ivscanf(id, “%lf%*t”, real_data)

‘  Display the response as string in a Message Box
   buf = Str$(real_data)
   retval = MsgBox(buf, MB_OK, “GPIO Data”)

‘  Close the device session.
   Call iclose(id)
```

```
‘  Enable the button used to initiate I/O
   cmdMeas.Enabled = True

   Exit Sub

ErrorHandler:
‘  Display the error message string in a Message Box
   retval = MsgBox(Error$, MB_ICONEXCLAMATION, “SICL Error”)

‘  Close the device session if iopen was successful.
   If id <> 0 Then
     iclose (id)
   End If

‘  Enable the button used to initiate I/O
   cmdMeas.Enabled = True
   Exit Sub

End Sub

‘  The following routine is called when the application’s
‘  Start Up form is unloaded. It calls siclcleanup to
‘  release resources allocated by SICL for this
‘  application. ‘

Sub Form_Unload (Cancel As Integer)
   Call siclcleanup ‘ Tell SICL to clean up for this task
End Sub
```

Example: GPIO Interrupts

```
/* gpiointr.c
   This program:
   - Creates a GPIO session with error checking
   - Installs an interrupt handler and enables EIR interrupts
   - Waits for EIR; invokes the handler for each interrupt
*/

#include <sicl.h>

void SICLCALLBACK handler(id, reason, sec)
INST id;
int reason, sec;
{
```

```
   if (reason == I_INTR_GPIO_EIR) {
      printf("EIR interrupt detected\n");

   /* Proper protocol is for the peripheral device to hold
    * EIR asserted until the controller "acknowledges" the
    * interrupt. The method for acknowledging and/or responding
    * to EIR is very device-dependent. Perhaps a CTLx line is
    * pulsed, or data is read, etc. The response should be
    * executed at this point in the program.
    */
   }
   else
      printf("Unexpected Interrupt; reason=%d\n", reason);
}

main()
{
   INST intf;     /* interface session id */

   #if defined (__BORLANDC__) && !defined (__WIN32__)
   _InitEasyWin(); /* required for Borland EasyWin programs */
   #endif

   /* log message and exit program on error */
   ionerror(I_ERROR_EXIT);

   /* open GPIO interface session */
   intf = iopen("gpio");

   /* suspend interrupts until configured */
   iintroff();

   /* configure interrupts */
   ionintr(intf, handler);
   isetintr(intf, I_INTR_GPIO_EIR, 1);

   /* wait for interrupts */
   printf("Ready for interrupts\n");
   while (1) {
      iwaithdlr(0);  /* optional timeout can be specified here*/
   }
```

```
   /* iwaithdlr performs an automatic iintron(). If your program
    * does concurrent processing, instead of waiting you need
    * to execute iintron() when you are ready for interrupts.
    */

    /* This simplified example loops forever. Most real applications
    * would have termination conditions that cause the loop to exit.
    */
   iclose(id);

   /* This call is a no-op for WIN32 applications. */

   _siclcleanup();
}
```

6

Using SICL with VXI

Using SICL with VXI

This chapter shows how to use SICL to communicate over the VXIbus. The example programs in this chapter are also provided in the **C\SAMPLES\MISC** subdirectory under the SICL base directory. This chapter includes:

- Introduction
- Using VXI Device Sessions
- Using VXI Interface Sessions
- Communicating with VME Devices
- SICL Function Support for VXI
- VXI Backplane Memory I/O Performance
- Using VXI-Specific Interrupts

Introduction

This section provides an introduction to using SICL with the VXI interface, including:

- Selecting a VXI Communications Session
- SICL VXI Functions

Selecting a VXI Communications Session

Before you begin programming your VXI system, ensure the system is set up and operating correctly. To begin programming a VXI system, you must first determine the type of communication session to be used. The two types of supported VXI communication sessions are:

- **Device Session**. A VXI device session allows direct access to a device regardless of the type of interface to which the device is connected.
- **Interface Session**. A VXI interface session allows direct, low-level control of the specified interface that provides full control of the activities on a given interface, such as VXI.

Device sessions are the recommended method for communicating while using SICL since they provide the highest level of programming, best overall performance, and best portability.

NOTE

Commander Sessions are *not* supported with VXI interfaces.

SICL VXI Functions

A summary of VXI-specific functions follows. Using these VXI interface specific functions means that the program cannot be used on other interfaces and, therefore, becomes less portable. These functions will work over a LAN-gatewayed session if the server supports the operation.

SICL VXI Functions

Function Name	Action
`ivxibusstatus`	Returns requested bus status information
`ivxigettrigroute`	Returns the routing of the requested trigger line
`ivxirminfo`	Returns information about VXI devices
`ivxiservants`	Identifies active servants
`ivxitrigoff`	De-asserts VXI trigger line(s)
`ivxitrigon`	Asserts VXI trigger line(s)
`ivxitrigroute`	Routes VXI trigger lines
`ivxiwaitnormop`	Suspends until normal operation is established
`ivxiws`	Sends a word-serial command to a device

Using VXI Device Sessions

This section gives guidelines to communicate directly with VXI devices using VXI **device sessions**.

VXI Device Types

There are two different types of VXI devices: **message-based** and **register-based**. To program a VXIbus system that is mixed with both message-based and register-based devices, open a communications session for each device in the system and program as shown in the following sections.

- **Message-Based Devices**. Message-based devices have their own processors that allow them to interpret high-level SCPI (Standard Commands for Programmable Instruments) commands. When using SICL, place the SCPI command within the SICL output function call and the message-based device interprets the SCPI command.

- **Register-Based Devices**. Register-based devices typically do not have their own processor to interpret high-level commands and therefore accept only binary data. You can use the following methods to program register-based devices:

  - **Interpreted SCPI**. Use the SICL **`iscpi`** interface and program using high-level SCPI commands. I-SCPI interprets high-level SCPI commands and sends the data to the instrument. Interpreted SCPI (I-SCPI) is supported over LAN, but register programming (**`imap`**, **`ipeek`**, **`ipoke`**, etc) is *not* supported over LAN. I-SCPI runs on a LAN server in a LAN-based system.

  - **Register programming.** Do register peeks and pokes and program directly to the device's registers with the **`vxi`** interface.

- **Compiled SCPI**. Use the C-SCPI product and program with high-level SCPI commands (achieve higher throughput as well).

- **Command Module**. Use a Command Module to interpret the high-level SCPI commands. The **`gpib`** interface is used with a Command Module. A Command Module may also be accessed over a LAN using a LAN-to-GPIB gateway.

Using VXI Message-Based Devices

Message-based devices have their own processors which allow them to interpret high-level SCPI commands. When using SICL, place the SCPI command within the SICL output function call and the message-based device interprets the SCPI command. SICL functions used for programming message-based devices include **`iread`**, **`iwrite`**, **`iprintf`**, **`iscanf`**, etc..

> **NOTE**
>
> If a message-based device has shared memory, you can access the device's shared memory with register peeks and pokes. See "Register-Based Devices" in this chapter for information on register programming.

Addressing VXI Message-Based Devices

To create a VXI device session, specify the interface symbolic name or logical unit and a device's address in the *addr* parameter of the **`iopen`** function. The interface symbolic name and logical unit are set by running the **`IO Config`** utility. To open **`IO Config`**, click the **`Agilent IO Libraries Control`** and then click **`Run IO Config.`** See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on **`IO Config`**.

Primary address must be between 0 and 255. The primary address corresponds to the VXI logical address and specifies the address in A16 space of the VXI device. SICL supports only primary addressing on the VXI device sessions. Specifying a secondary address causes an error.

Some example addresses for VXI device sessions follow. These examples use the default symbolic name specified during the system configuration. To change the name listed, you must also change the symbolic name or logical unit specified during the configuration. The name used in the SICL program must match the logical unit or symbolic name specified in the system configuration. Other possible interface names are **`VXI`**, **`vxi`**, etc..

`vxi,24`	A device address corresponding to the device at primary address 24 on the vxi interface.
`vxi,128`	A device address corresponding to the device at primary address 128 on the vxi interface.

An example of opening a device session with the VXI device at logical address 64 follows.

```
INST dmm;
dmm = iopen (“vxi,64”);
```

Example: VXI Message-Based Device Session (C)

This example program opens a communication session with a VXI message-based device and measures the AC voltage. The measurement results are then printed.

```
/* vximdev.c
   This example program measures AC voltage on a
   multimeter andprints out the results */

#include <sicl.h>
#include <stdio.h>

main()
{
   INST dvm;
   char strres[20];

   /* Print message and terminate on error */
   ionerror (I_ERROR_EXIT);

   /* Open the multimeter session */
   dvm = iopen (“vxi,24”);
   itimeout (dvm, 10000);

   /* Initialize dvm */
   iwrite (dvm, “*RST\n”, 5, 1, NULL);

   /* Take measurement */
   iwrite (dvm,”MEAS:VOLT:AC? 1, 0.001\n”, 23, 1, NULL);

   /* Read measurements */
   iread (dvm, strres, 20, NULL, NULL);

   /* Print the results */
   printf(“Result is %s\n”, strres);

   /* Close the multimeter session */
iclose(dvm);
}
```

Using Register-Based Devices

Communication Methods

Several methods you can use to communicate with register-based devices follow. For a SICL application that accesses VXI devices using GPIB and a Command Module, you can port your application to use the **`iscpi`** interface and directly access the VXI backplane without the use of the Command Module. Do this by changing the **`iopen`** function to use the **`iscpi`** interface followed by the device's logical address.

See "Addressing VXI Register-Based Devices" in this chapter for more details on addressing rules. Since I-SCPI was designed to simulate control of register-based instruments using GPIB and the Command Module, you usually will not need to change anything else in your application.

There are other applications that use SICL as their I/O library, but have their own methods of communicating with the instruments. These applications hide most of the I/O complexity behind the user interface. Contact your local sales representative for information on other products that might interpret the high-level SCPI commands for register-based devices.

- **`iscpi` interface**. Use the SICL **`iscpi`** interface and program using SCPI commands. The **`iscpi`** interface interprets the SCPI commands and allows direct communication with register-based devices.This method is supported over LAN.

NOTE

Agilent VISA must be installed to use the **`iscpi`** interface.

- **Register Programming**. Use the **`vxi`** interface to program directly to the device's registers with a series of register peeks and pokes. This method can be very time-consuming and difficult. This method is not supported over LAN.

- **Compiled SCPI**. Compiled SCPI product is a programming language that can be used with SICL to program register-based devices using SCPI commands. Because Compiled SCPI interprets SCPI commands at compile time, Compiled SCPI can be used to achieve high throughput of register-based devices.

- **Command Module**. You can use a Command Module to communicate with VXI devices via GPIB. The Command Module interprets the high-level SCPI commands for register-based instruments and sends low-level commands over the VXIbus backplane to the instruments. See *Chapter 4 - Using SICL with GPIB* for details on communicating via a Command Module.

Addressing VXI Register-Based Devices

To create a device session, specify the interface symbolic name or logical unit and a device's address in the *addr* parameter of the **`iopen`** function. The interface symbolic name and logical unit are set by running the **`IO Config`** utility. To open **`IO Config`**, click the **`Agilent IO Libraries Control`** and then click **`Run IO Config.`** See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on **`IO Config`**.

The primary address corresponds to the VXI logical address and must be between 0 and 255. SICL supports only primary addressing on VXI device sessions. Specifying a secondary address causes an error. Some example addresses for VXI device sessions follow.

These examples use the default symbolic name specified during the system configuration. To change the name listed, you must also change the symbolic name or logical unit specified during the configuration. The name used in your SICL program must match the logical unit or symbolic name specified in the system configuration. Other possible interface names are **`VXI`**, **`vxi`**, etc..

`iscpi,32`	A register-based device address corresponding to the device at primary address 32 on the iscpi interface.
`vxi,24`	A device address corresponding to the device at primary address 24 on the vxi interface.
`vxi,128`	A device address corresponding to the device at primary address 128 on the vxi interface.

An example of opening a device session with the VXI device at logical address 64 follows.

```
INST dmm;
dmm = iopen (“vxi,64”);
```

Interpreted SCPI (`iscpi`) Addressing Rules

The simplest way to address a register-based device using the Interpreted SCPI (I-SCPI or **`iscpi)`** interface is to specify the interface logical unit or symbolic name and a device logical address in the *addr* parameter of the **`iopen`** function. For example:

```
dmm=iopen (“iscpi,24”);
```

I-SCPI automatically configures your system according to combining rules that determine how instruments are set up relative to other VXI instruments.

Generally, when an **`iopen`** is performed, an **instrument** is formed consisting of all devices at logical addresses contiguous to the base logical address passed in the address string. For example, if you open an instrument at logical address 24 with the next logical address at 25, the **`iscpi`** interface searches for an instrument driver that supports the devices found.

Defining an Instrument. For control of logical addresses used to form a particular instrument, you can use an explicit list in the logical address portion of the **`iopen`** call. Define the instrument by adding a colon after the interface symbolic name, followed by the backplane name as specified in the **`IO Config`** utility (backplane is the *symname* of the VXI backplane SICL driver, usually **`vxi`**). Then, add the instrument logical addresses enclosed within parentheses separated by commas.

This example combines instruments at logical address 24 and 25 to form one instrument. The logical addresses of these instruments do not have to be contiguous.

```
dmm=iopen (“iscpi:vxi,(24,25)”);
```

Defining an Instrument Driver. To specify an instrument driver to use for a specific set of logical addresses, add the instrument driver name within brackets. This allows you to create your own instrument drivers or you can form unique virtual instrument combinations. For example:

```
dmm=iopen (“iscpi,24[E1326]”);
```

To specify an instrument driver plus the instruments grouped together to form the instrument, use the following form. The **`iopen`** call will run faster if you specify an instrument driver name since it does not have to search through all the instrument drivers for a match.

```
dmm=iopen (“iscpi[E1326]:vxi,(24,25)”);
```

The directory location specified during the SICL installation is searched for a matching instrument driver.

Programming with Interpreted SCPI (the `iscpi` Interface)

The Interpreted SCPI (I-SCPI or **`iscpi)`** interface allows you to program register-based instruments with high-level SCPI commands. To program using the **`iscpi`** interface, open a device session with a specific register-based instrument and then program using the SICL functions such as **`iprintf`**, **`iscanf`**, and **`ireadstb`**.

To use the **`iscpi`** interface, you must first have configured the system with the **`IO Config`** utility to include **`iscpi`** as an interface. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on **`IO Config`**.

When opening the device session, you will need to specify **`iscpi`** as the interface type in the SICL **`iopen`** call. See "Interpreted SCPI (**`iscpi`**) Addressing Rules" in this chapter for information on addressing with the **`iscpi`** interface.

The **`iscpi`** interface was designed to closely simulate control of register-based instruments using a Command Module via GPIB. When an **`iopen`** is performed, I-SCPI searches for an instrument driver consisting of all the devices at logical addresses contiguous to the base logical address.

If no instrument driver supports the list of contiguous logical addresses, the device with the highest logical address will be removed and the search process repeated. This continues until the driver is found or this list is exhausted. If no instrument driver is found, the **`iopen`** call will fail.

Once an **`iopen`** is successful, I-SCPI runs in an infinite loop waiting to parse SCPI commands for the instrument. A separate process is created for each instrument that is opened.

Register-Based Instrument Drivers. The **`iscpi`** interface includes drivers for most Agilent register-based devices. These drivers are located in the VISA directory specified during the Agilent IO Libraries installation (default is **`C:\Program Files\VISA\WIN95\BIN`** (Windows 95/98) or **`C:\Program Files\VISA\WINNT\BIN`** (Windows NT/2000). See the **`C:\Program Files\VISA\WINxx\BIN\iscpinfo.TXT`** file for a list of currently supported register-based devices.

Example: `iscpi` Device Session

This example program opens a communication session with a VXI register-based device with the **`iscpi`** interface and then uses SCPI commands to measure the AC voltage and print out the results.

```
/* vxiiscpi.c
   This example program measures AC voltage on a
   multimeter and prints out the results */

#include <sicl.h>
#include <stdio.h>

main()
{
   INST dvm;
   char strres[20];

   /* Print message and terminate on error */
   ionerror (I_ERROR_EXIT);

   /* Open the multimeter session */
   dvm = iopen (“iscpi,24”);
   itimeout (dvm, 10000);

   /* Initialize dvm */
   iwrite (dvm, “*RST\n”, 5, 1, NULL);

   /* Take measurement */
   iwrite (dvm,”MEAS:VOLT:AC? 1, 0.001\n”, 23, 1, NULL);

   /* Read measurements */
   iread (dvm, strres, 20, NULL, NULL);

   /* Print the results */
   printf(“Result is %s\n”, strres);

   /* Close the multimeter session */
   iclose(dvm);
}
```

Programming Directly to the Registers

When communicating with register-based devices, you must either send a series of peeks and pokes directly to the device's registers or use a command interpreter to interpret the high-level SCPI commands. Command interpreters include the **`iscpi`** interface, Agilent Command Module, Agilent B-Size Mainframe (built-in Command Module), or Compiled SCPI (C-SCPI).

When sending a series of peeks and pokes to the device's registers, use the following process. This procedure is only used on register-based devices that are not using the **`iscpi`** interface. Note that programming directly to the registers is not supported over LAN.

- Map memory space into your process space.
- Read the register's contents using **`i?peek`**.
- Write to the device registers using **`i?poke`**.
- Unmap the memory space.

Mapping Memory Space for Register-Based Devices

When using SICL to communicate directly to the device's registers, you must map a memory space into the process space by using the SICL **`imap`** function:

```
imap (id, map_space, pagestart, pagecnt, suggested);
```

This function maps space for the interface or device specified by the *id* parameter. *pagestart*, *pagecnt*, and *suggested* indicate the page number, numbesr of pages, and a suggested starting location respectively. *map_space* determines which memory location to map the space.

Due to hardware constraints on given devices or interfaces, 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`** 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. Some Valid *map_space* choices follow.

.

Function	Description
I_MAP_A16	Maps in VXI A16 address space (device or interface sessions, 64K byte pages).
I_MAP_A24	Maps in VXI A24 address space (device or interface sessions, 64K byte pages).
I_MAP_A32	Maps in VXI A32 address space (device or interface sessions, 64K byte pages).
I_MAP_VXIDEV	Maps in VXI A16 device registers (device session only, 64 bytes).
I_MAP_EXTEND	Maps in VXI device extended memory address space in A24 or A32 address space (device sessions only).
I_MAP_SHARED	Maps in VXI A24/A32 memory that is physically located on the computer (sometimes called local shared memory, interface sessions only).
I_MAP_AM | *address modifer*	Maps in the specified region (*address modifer*) of VME address space. See the “Communicating with VME Devices” section later in this chapter for more information on this map space argument

Some example **imap** function calls follow.

```
/* Map to the VXI device vm starting at pagenumber 0
for 1 page */
base_address = imap (vm, I_MAP_VXIDEV, 0, 1, NULL);

/* Map to A32 address space (16 Mbytes) */
ptr = imap (id, I_MAP_A32, 0x000, 0x100, NULL);

/* Map to a device’s A24 or A32 extended memory */
ptr=imap (id, I_MAP_EXTEND, 0, 1, 0);

/* Map to a computer’s A24 or A32 shared memory */
ptr=imap (id, I_MAP_SHARED, 0, 1, 0);
```

Use the following table to determine which *map-space* argument to use with a SICL **imap/iunmap** function. All accesses through the *_D32 map windows can *only* be 32-bit transfers. The application software must do a 32-bit assignment to generate the access and only accesses on 32-bit boundaries are allowed. If 8- or 16-bit accesses to the device are also necessary, a normal **I_MAP_A16/24/32** map must also be requested.

imap/iunmap (*map-space* argument)	**Widths**	**VME Data Access Mode**
I_MAP_A16	D8,D16	Supervisory
I_MAP_A24	D8,D16	Supervisory
I_MAP_A32	D8,D16	Supervisory
I_MAP_A16_D32	D32	Supervisory
I_MAP_A24_D32	D32	Supervisory
I_MAP_A32_D32	D32	Supervisory

Reading and Writing to Device Registers

When you have mapped the memory space, use the SICL **i?peek** and **i?poke** functions to communicate with register-based instruments. With these functions, you need to know which register you want to communicate with and the register's offset. See the instrument's user's manual for a description of the registers and register locations. See *Chapter 11 - SICL Language Reference* for a description of the **i?peek** and **i?poke** functions. An example using **iwpeek** follows.

```
id = iopen ("vxi,24");
addr = imap (id, I_MAP_VXIDEV, 0, 1, 0);
reg_data = iwpeek (addr + 4);
```

Unmapping Memory Space

Be sure you use the **iunmap** function to unmap the memory space when the space is no longer needed. This frees the mapping hardware so it can be used by other processes.

Example: VXI Register-Based Programming (C)

This example program opens a communication session with a register-based device connected to the address entered by the user. The program then reads the **Id** and **Device Type** registers and the prints the register contents.

```
/* vxirdev.c
The following example prompts the user for an instrument
address and then reads the id register and device type
register. The contents of the register are displayed.*/

#include <stdio.h>
#include <stdlib.h>
#include <sicl.h>

void main (){
  char inst_addr[80];
  char *base_addr;
  unsigned short id_reg, devtype_reg;
  INST id;

  /* get instrument address */
  puts ("Please enter the logical address of the
         register-based instrument, for example,
         vxi,24 :  \n");
  gets (inst_addr);

  /* install error handler */
  ionerror (I_ERROR_EXIT);

  /* open communications session with instrument */
  id  =  iopen (inst_addr);
  itimeout (id, 10000);

  /* map into user memory space */
  base_addr = imap (id, I_MAP_VXIDEV, 0, 1, NULL);

  /* read registers */
  id_reg = iwpeek ((unsigned short *)(base_addr + 0x00));
  devtype_reg = iwpeek ((unsigned short *)(base_addr + 0x02));

  /* print results */
  printf ("Instrument at address %s\n", inst_addr); printf
         "ID Register = 0x%4X\n  Device Type Register =
         0x%4X\n", id_reg, devtype_reg);
  /* unmap memory space */
  iunmap (id, base_addr, I_MAP_VXIDEV, 0, 1);

 /* close session */
  iclose (id);}
```

Using VXI Interface Sessions

VXI **interface sessions** allow direct low-level control of the interface. However, the programmer must provide all bus maintenance for the interface and have considerable knowledge of the interface. When using interface sessions, you must use interface-specific functions which means the program cannot be used on other interfaces and becomes less portable.

Addressing VXI Interface Sessions

To create an interface session on a VXI system, specify the interface symbolic name or logical unit in the *addr* parameter of the **iopen** function. The interface symbolic name and logical unit are set by running the **IO Config** utility. To open **IO Config**, click the **Agilent IO Libraries Control** and then click **Run IO Config.** See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on **IO Config**.

Some example addresses for VXI interface sessions follow. These examples use the default symbolic name specified during the system configuration. To change the name listed, you must also change the symbolic name or logical unit specified during the configuration.

The name used in your SICL program must match the logical unit or symbolic name specified in the system configuration. Other possible interface names are **VXI**, **vxi**, etc. The only interface session operations supported by I-SCPI are service requests and locking.

vxi	An interface symbolic name.
iscpi	An interface symbolic name.

This example opens a interface session with the VXI interface.

```
INST vxi;
vxi = iopen (“vxi”);
```

Example: VXI Interface Session (C)

This example program opens a communication session with the VXI interface and uses the SICL interface specific **ivxirminfo** function to get information about a specific VXI device. This information comes from the VXI resource manager and is only valid as of the last time the VXI resource manager was run.

```
 /* vxiintr.c
   The following example gets information about a specific
   vxi device and prints it out. */
#include <stdio.h>
#include <sicl.h>

void main () {
   int laddr;
   struct vxiinfo info;
   INST id;

   /* get instrument logical address */
   printf ("Please enter the logical address of the
           register-based instrument, for example,
           24 : \n");
   scanf ("%d", &laddr);

   /* install error handler */
   ionerror (I_ERROR_EXIT);

   /* open a vxi interface session */
   id  =  iopen ("vxi");
   itimeout (id, 10000);

   /*read VXI resource manager information for
     specified device*/
   ivxirminfo (id, laddr, &info);

   /* print results */
   printf ("Instrument at address %d\n", laddr);
   printf ("Manufacturer's Id = %s\n  Model = %s\n",
            info.manuf_name, info.model_name);

  /* close session */
   iclose (id);
}
```

Communicating with VME Devices

Although VXI is an extension of VME, VME is not easy to use in a VXI system. Since the VXI standard defines specific functionality that would be custom designs in VME, some resources required for VME custom design are actually used by VXI. Therefore, there are certain limitations and requirements when using VME in a VXI system.

NOTE

VME is not an officially supported interface for SICL and is not supported over LAN.

Use these process when using VME devices in a VXI mainframe:

- Declaring Resources
- Mapping VME Memory
- Reading and Writing to Device Registers
- Unmapping Memory

Declaring Resources

The VXI Resource Manager does not reserve resources for VME devices. Instead, a configuration file is used to reserve resources for VME devices in a VXI system. Use the VXI Device Configurator to edit the **`DEVICES`** file (or edit the file directly) to reserve resources for VME devices. The VXI Resource Manager reads this file to reserve the VME address space and VME IRQ lines. The VXI Resource Manager then assigns the VXI devices around the already reserved VME resources.

For VME devices requiring A16 address space, the device's address space should be defined in the lower 75% of A16 address space (addresses below 0xC000). This is necessary because the upper 25% of A16 address space is reserved for VXI devices.

For VME devices using A24 or A32 address space, use A24 or A32 address ranges just higher than those used by your VXI devices. This will prevent the VXI Resource Manager from assigning the address range used by the VME device to any VXI device. (The A24 and A32 address range is software programmable for VXI devices.)

Mapping VME Memory

SICL defaults to byte, word, and longword supervisory access to simplify programming VXI systems. However, some VME cards use other modes of access that are not supported in SICL. Therefore, SICL provides a map parameter that allows you to use the access modes defined in the *VMEbus Specification*. See the *VMEbus Specification* for information on these access modes.

NOTE

Use care when mixing VXI and VME devices. You *must* know the VME address space and offset within that address space the VME devices use. VME devices cannot use the upper 16K of the A16 address space since this area is reserved for VXI instruments.

When accessing VME or VXI devices via an embedded controller, current versions of SICL use the “supervisory data” address modifiers 0x2D, 0x3D, and 0x0D for A16, A24, and A32 accesses, respectively. (Some older versions of SICL use the “non-privileged data” address modifiers.)

Use the **`I_MAP_AM`** | *address modifer* map space argument in the **`imap`** function to specify the map space region (*address modifer*) of VME address space. See the *VMEbus Specifications* for information on values to use as the address modifier. If the controller does not support specified address mode, the **`imap`** call will fail (see table in the next section).

This maps A24 non-privileged data access mode:

```
prt = imap (id, (I_MAP_AM | 0x39), 0x20, 0x4, 0);
```

This maps A32 non-privileged data access mode:

```
prt = imap (id, (I_MAP_AM | 0x09), 0x20, 0x40, 0);
```

This table lists VME access modes supported on Hewlett-Packard controllers.

VME Mapping Support

	A16			A24			A32		
	D08	D16	D32	D08	D16	D32	D08	D16	D32
Supervisory data	X	X	X	X	X	X	X	X	X
Non-Privileged data									

Reading and Writing to Device Registers

After you have mapped the memory space, use the SICL **i?peek** and **i?poke** functions to communicate with the VME devices. With these functions, you need to know the register to communicate with and the register's offset.

See the instrument's user's manual for descriptions of registers and register locations. See *Chapter 11 - SICL Language Reference* for a description of the **i?peek** and **i?poke** functions. This is an example using **iwpeek**:

```
id = iopen ("vxi");
addr = imap (id, (I_MAP_AM | 0x39), 0x20, 0x4, 0);
reg_data = iwpeek ((unsigned short *)(addr + 0x00));
```

Unmapping Memory Space

Make sure you use the **iunmap** function to unmap the memory space when it is no longer needed. This frees the mapping hardware so it can be used by other processes.

VME Interrupts

There are seven VME interrupt lines that can be used. By default, VXI processing of the IACK value will be used. However, if you configure VME IRQ lines and **VME Only**, no VXI processing of the IACK value will be done. That is, the IACK value will be passed to a SICL interrupt handler directly. See **isetintr** in *Chapter 11 - SICL Language Reference* for information on the VME interrupts.

Example: VME Interrupts (C)

This ANSI C example program opens a VXI interface session and sets up an interrupt handler. When the **I_INTR_VME_IRQ1** interrupt occurs, the function defined in the interrupt handler will be called. The program then writes to the registers, causing the **I_INTR_VME_IRQ1** interrupt to occur.

You must edit this program to specify the starting address and register offset of your specific VME device. This example program also requires the VME device to be using **I_INTR_VME_IRQ1** and the controller to be the handler for the VME IRQ1.

```
/* vmedev.c
 This example program opens a VXI interface session and sets
 up an interrupt handler. When the specified interrupt occurs,
 the procedure defined in the interrupt handler is called. You
 must edit this program to specify starting address and
 register offset for your specific VME device. */

#include <stdio.h>
#include <stdlib.h>
#include <sicl.h>

#define ADDR “vxi”

void handler (INST id, long reason, long secval){
   printf (“Got the interrupt\n”);
}

void main ()
{
   unsigned short reg;
   char *base_addr;
   INST id;

   /* install error handler */
   ionerror (I_ERROR_EXIT);

   /* open an interface communications session */
   id = iopen (ADDR);
   itimeout (id, 10000);

   /* install interrupt handler */
   ionintr (id, handler);
   isetintr (id, I_INTR_VME_IRQ1, 1);

   /* turn interrupt notification off so that interrupts are
      not recognized before the iwaithdlr function is called*/
   iintroff ();

   /* map into user memory space */
   base_addr = imap (id, I_MAP_A24, 0x40, 1, NULL);

   /* read a register */
   reg = iwpeek((unsigned short *)(base_addr + 0x00));
```

```
  /* print results */
  printf ("The registers contents were as follows:
           0x%4X\n", reg);

  /* write to a register causing interrupt */
  iwpoke ((unsigned short *)(base_addr + 0x00), reg);

  /* wait for interrupt */
  iwaithdlr (10000);

  /* turn interrupt notification on */
  iintron ();

  /* unmap memory space */
  iunmap (id, base_addr, I_MAP_A24, 0x40, 1);

  /* close session */
  iclose (id);
}
```

SICL Function Support with VXI

This section describes how SICL functions are implemented for VXI **device sessions** and **interface sessions**.

VXI Message-Based Device Sessions

This section describes how some SICL functions are implemented for VXI device sessions for message-based devices.

`iwrite`	Sends the data to the (message-based) servant using the byte-serial write protocol and the *byte available* word-serial command.
`iread`	Reads the data from the (message-based) servant using the byte-serial read protocol and the *byte request* word-serial command.
`ireadstb`	(read status byte) Performs a VXI *readSTB* word-serial command.
`itrigger`	Sends a word-serial *trigger* to the specified message-based device.
`iclear`	Sends a word-serial *device clear* to the specified message-based device.
`ionsrq`	Can be used to catch SRQs from message-based devices.

Interpreted SCPI Device Sessions

The **iscpi** interface is used to program VXI register-based instruments. However, the VXI specific and register-based specific SICL functions such as **ivxiws**, **imap**, and **ipeek** are not necessary and are not implemented for the **iscpi** interface. The following describes how some SICL functions are implemented for **iscpi** device sessions.

iwrite	Sends the SCPI commands to the register-based instrument driver's input buffer. The driver will interpret the command and do register peeks and pokes. If the command is a query, the driver will put the data into its output buffer.
iread	Reads the data from the register-based instrument driver's output buffer.
ireadstb	Performs the equivalent of a serial poll (SPOLL).
itrigger	Performs the equivalent of an addressed group execute trigger (GET).
iclear	Performs the equivalent of a device clear (DCL) on the device corresponding to this session.

Interpreted SCPI Device Sessions Interrupts

The **iscpi** interface does not support interrupts, so the SICL **ionintr** function is not implemented for **iscpi** device sessions. There are no device-specific interrupts for the **iscpi** interface.

Interpreted SCPI Device Sessions Service Requests

iscpi device sessions support Service Requests (SRQ) in the same manner as GPIB. When one device issues an SRQ, *all* **iscpi** device sessions that have SRQ handlers installed (see **ionsrq** in *Chapter 11 - SICL Language Reference*) will be informed. This is an emulation of how GPIB handles the SRQ line.

The interface cannot distinguish which device requested service, so **iscpi** acts as if all devices require service. Your SRQ handler can retrieve the device's **status byte** by using the **ireadstb** function. The status byte can be used to determine if the instrument needs service. It is good practice to ensure that a device is not requesting service before leaving the SRQ handler. The easiest technique for this is to service all devices from one handler.

VXI Register-Based Device Sessions

Because VXI *register-based* devices do not support the word serial protocol and other features of *message-based* devices, the following SICL functions are **not** supported with register-based device sessions unless you use the **`iscpi`** interface.

All other functions will work with all VXI devices (message-based, register-based, etc.). Use the **`i?peek`** and **`i?poke`** functions to communicate with register-based devices.

Category	Functions Not Supported
Non-formatted I/O	**`iread, iwrite, itermchr`**
Formatted I/O	**`iprintf, iscanf, ipromptf, ifread, ifwrite, iflush, isetbuf, isetubuf`**
Device/Interface Control	**`iclear, ireadstb, isetstb, itrigger`**
Service Requests	**`igetonsrq, ionsrq`**
Timeouts	**`igettimeout, itimeout`**
VXI Specific	**`ivxiws`**

VXI Interface Sessions

The following describes how some SICL functions are implemented for VXI interface sessions. I-SCPI interface sessions only support service requests and locking (**`ionsrq`**, **`ilock`**, and **`iunlock`**).

`iwrite and iread`	Not supported for VXI interface sessions. Returns the `I_ERR_NOTSUPP` error.
`iclear`	Causes the VXI interface to perform a SYSREST on interface sessions. This causes all VXI devices to reset. If the `iscpi` interface is being used, the `iscpi` instrument will be terminated. If this happens, a `No Connect` error message occurs and you must reopen the `iscpi` communications session. All servant devices cease to function until the VXI resource manager runs and normal operation is re-established.

VXI Backplane Memory I/O Performance

SICL supports two different memory I/O mechanisms for accessing memory on the VXI backplane.

Single location peek/ poke and direct memory dereference	**`imap, iunmap, ibpeek, iwpeek, ilpeek, ibpoke, iwpoke, ilpoke,`** *value = *pointer, *pointer = value*
Block memory access	**`imap, iunmap, ibblockcopy, iwblockcopy, ilblockcopy, ibpushfifo, iwpushfifo, ilpushfifo ibpopfifo, iwpopfifo, ilpopfifo`**

Using Single Location Peek/ Poke

Single location peek/poke or direct memory dereference is the most efficient in programs that require repeated access to different addresses. On many platforms, the peek/poke operations are actually macros which expand to direct memory dereferencing.

An exception is Windows platforms, where **`ipeek`**/**`ipoke`** are implemented as functions since (under certain conditions) the compiler will attempt to optimize a direct dereference and cause a VXI memory access of the wrong size. For example, when masking the results of a 16-bit read in an expression:

```
data = iwpeek(addr) & 0xff;
```

the compiler will simplify this to an 8-bit read of the contents of the **`addr`** pointer. This would cause an error when attempting to read memory on a VXI card that did not support 8-bit access. When **`iwpeek`** is implemented as a function, the correct size memory access is guaranteed.

Using Block Memory Access

The block memory access functions provide the highest possible performance for transferring large blocks of data to or from the VXI backplane. Although these calls have higher initial overhead than the **`ipeek`**/**`ipoke`** calls, they are optimized on each platform to provide the fastest possible transfer rate for large blocks of data.

These routines may use DMA, which is not available with **`ipeek`**/**`ipoke`**. For small blocks, the overhead associated with the block memory access functions may actually make these calls longer than an equivalent loop of **`ipeek`**/**`ipoke`** calls.

The block size at which the block functions become faster depends on the particular platform and processor speed.

Example: VXI Memory I/O (C)

An example follows that demonstrates the use of simple and block memory I/O methods in SICL.

```
/*
   siclmem.c
   This example program demonstrates the use of
   simple and block memory I/O methods in SICL. */

#include <sicl.h>
#include <stdlib.h>
#include <stdio.h>

#define VXI_INST "vxi,24"

void main () {
   INST           id;
   unsigned short *memPtr16;
   unsigned short id_reg;
   unsigned short devtype_reg;
   unsigned short memArray[2];
   int            err;

   /* Open a session to the instrument */
   id = iopen(VXI_INST);

/* ============== Simple memory I/O =================
   = iwpeek()
   = direct memory dereference

   On many platforms, the ipeek/ipoke operations are actually
   macros which expand to direct memory dereferencing. The
   exception is on Microsoft Windows platforms where ipeek/
   ipoke are implemented as functions.

   This is necessary because under certain conditions, the
   compiler will attempt to optimize a direct dereference and
   cause a VXI memory access of the wrong size. For example,
   when masking the results of a 16-bit read in a expression:

      data = iwpeek(addr) & 0xff;
```

```
   the compiler will simplify this to an 8-bit read of the
   contents of the addr pointer. This would cause an error when
   attempting to read memory on a VXI card that did not support
   8-bit access. */

/* Map into memory space */
memPtr16 = (unsigned short *)imap(id, I_MAP_VXIDEV, 0, 1, 0);

/* ============ Using Peek ================== */

/* Read instrument id register contents */
id_reg = iwpeek(memPtr16);

/*       Read device type register contents    */
id_reg = iwpeek(memPtr16+1);

/* Print results */
printf("   iwpeek: ID Register = 0x%4X\n", id_reg);
printf("   iwpeek: Device Type Register = 0x%4X\n",
    devtype_reg);

/* Use direct memory dereferencing */
id_reg =      *memPtr16;
devtype_reg = *(memPtr16+1);

/* Print results */
printf("dereference: ID Register = 0x%4X\n", id_reg);
printf("dereference: Device Type Register = 0x%4X\n",
        devtype_reg);

/* ================ Block Memory I/O ==============
   = iwblockcopy
   = iwpushfifo
   = iwpopfifo

   These commands offer the best performance for reading and
   writing large data blocks on the VXI backplane. For this
   example, we are only moving 2 words at a time. Normally,
   these functions would be used to move much larger blocks
   of data. */
```

```
/* ============== Demonstrate Block Read =========== */

/* Read the instrument id register and device type
   register into an array. */

err = iwblockcopy(id, memPtr16, memArray, 2, 0);

/* Print results */
printf(" iwblockcopy: ID Register = 0x%4X\n", memArray[0]);
printf(" iwblockcopy: Device Type Register = 0x%4X\n",
memArray[1]);

/* ============ Demonstrate popfifo ================*/

/* Do a popfifo of the Id Register */
err = iwpopfifo(id, memPtr16, memArray, 2, 0);

/* Print results */
printf(" iwpopfifo: 1 ID Register = 0x%4X\n", memArray[0]);
printf(" iwpopfifo: 2 ID Register = 0x%4X\n", memArray[1]);

/* ================= Cleanup and Exit ==============*/

/* Unmap memory space */
iunmap(id, (char *)memPtr16, I_MAP_VXIDEV, 0, 1);

/* Close instrument session */
iclose(id);
}
```

Using VXI-Specific Interrupts

See the **`isetintr`** function in *Chapter 11 - SICL Language Reference* for a list of VXI specific interrupts.

Example: VXI Interrupt Actions (C)

This pseudo-code describes the actions performed by SICL when a VME interrupt arrives and/or a VXI signal register write occurs.

```
VME Interrupt arrives:
   get iack value
   send I_INTR_VME_IRQ?
   is VME IRQ line configured VME only
   if yes then
      exit
   do lower 8 bits match logical address of one of our servants?
   if yes then
      /*  iack is from one of our servants */
      call servant_signal_processing(iack)
   else
      /* iack is from non-servant VXI or VME device*/
   send I_INTR_VXI_VME interrupt to interface sessions

Signal Register Write occurs:
   get value written to signal register
   send I_INTR_ANY_SIG
   do lower 8 bits match logical address of one of our servants?
   if yes then
      /* Signal is from one of our servants */
      call Servant_signal_processing(value)
   else
      /* Stray signal */
      send I_INTR_VXI_UKNSIG to interface sessions
servant_signal_processing (signal_value)
   /* Value is form one of our servants */
   is signal value a response signal?
   If yes then
      process response signal
      exit
   /* Signal is an event signal */
   is signal an RT or RF event?
   if yes then
      /* A request TRUE or request FALSE arrived */
```

```
      process request TRUE or request FALSE event
      generate SRQ if appropriate
      exit
   is signal an undefined command event?
   if yes then
      /* Undefined command event */
      process an undefined command event
      exit
   /* Signal is a user-defined or undefined event */
   send I_INTR_VXI_SIGNAL to device sessions for this device
   exit
```

Example: Processing VME Interrupts (C)

```
/* vmeintr.c
   This example uses SICL to cause a VME interrupt from
   an E1361 register-based relay card at logical address 136.*/

#include <sicl.h>

static void vmeint (INST, unsigned short);
static void int_setup (INST, unsigned long);
static void int_hndlr (INST, long, long);
int intr = 0;
main() {
   int o;   INST id_intf1;
   unsigned long mask = 1;

   ionerror (I_ERROR_EXIT);
   iintroff ();
   id_intf1 = iopen ("vxi,136");
   int_setup (id_intf1, mask);
   vmeint (id_intf1, 136);
   /* wait for SRQ or interrupt condition */
   iwaithdlr (0);

   iintron ();
   iclose (id_intf1);
}
static void int_setup(INST id, unsigned long mask) {
   ionintr(id, int_hndlr);
   isetintr(id, I_INTR_VXI_SIGNAL, mask);
}
static void vmeint (INST id, unsigned short laddr) {
   int reg;
```

```
   char *a16_ptr = 0;

   reg = 8;
   a16_ptr = imap (id, I_MAP_A16, 0, 1, 0);
   /* Cause uhf mux to interrupt: */
   iwpoke ((unsigned short *)(a16_ptr + 0xc000 + laddr *
           64 + reg),  0x0);
}
static void int_hndlr (INST id, long reason, long sec) {
   printf ("VME interrupt: reason: 0x%x, sec: 0x%x\n",
              reason,sec);
   intr = 1;
}
```

Notes:

7

Using SICL with RS-232

Using SICL with RS-232

This chapter shows how to open a communications session and communicate with a device via an RS-232 connection. The example programs in this chapter are also provided in the **C\SAMPLES\MISC** (for C/C++) and **VB\SAMPLES\MISC** (for Visual Basic). The chapter includes:

- Introduction
- Using RS-232 Device Sessions
- Using RS-232 Interface Sessions

Introduction

This section provides an introduction to using SICL with the RS-232 interface, including:

- Selecting an RS-232 Communications Session
- SICL RS-232 Functions

Selecting an RS-232 Communications Session

RS-232 is a serial interface that is widely used for instrumentation. Although RS-232 is slow in comparison to GPIB or VXI, its low cost makes it an attractive solution in many situations. Because SICL for Windows uses the RS-232 facilities built into the Windows operating system, controlling RS-232 instruments is easy.

After you have configured your system for RS-232 communications, you can start programming using the SICL functions. Using SICL to communicate with a device via RS-232 is similar to using SICL to communicate via the GPIB interface. To use SICL, you must first determine the type of communications session required. An RS-232 communications session can be either a **device session** or an **interface session**. Commander sessions are not supported on RS-232.

Device Sessons. For direct access to a device, communication is with a device session. An RS-232 device session should be used when sending commands and receiving data from an instrument.

Interface Sessions. SICL also allows interface-specific actions, such as setting device addresses or other interface-specific characteristics. To do this, you communicate with an interface session. Setting interface characteristics (such as the baud rate) must be done with an interface session.

With RS-232, only one device is connected to the interface, so it may seem like extra work to have both device sessions and interface sessions. However, structuring the code so that interface-specific actions are isolated from actions on the device itself makes programs easier to maintain. This is especially important if you want to use a program with a similar device on a different interface, such as GPIB.

SICL RS-232 Functions

Function Name	Action
`iserialctrl`	Sets the following characteristics of the RS-232 interface:

Request	Characteristic	Settings
`I_SERIAL_BAUD`	Data rate	2400, 9600, etc.
`I_SERIAL_PARITY`	Parity	**`I_SERIAL_PAR_NONE`** **`I_SERIAL_PAR_IGNORE`** **`I_SERIAL_PAR_EVEN`** **`I_SERIAL_PAR_ODD`** **`I_SERIAL_PAR_MARK`** **`I_SERIAL_PAR_SPACE`**
`I_SERIAL_STOP`	Stop bits / frame	**`I_SERIAL_STOP_1`** **`I_SERIAL_STOP_2`**
`I_SERIAL_WIDTH`	Data bits / frame	**`I_SERIAL_CHAR_5`** **`I_SERIAL_CHAR_6`** **`I_SERIAL_CHAR_7`** **`I_SERIAL_CHAR_8`**
`I_SERIAL_READ_BUFSZ`	Receive buffer size	Number of bytes
`I_SERIAL_DUPLEX`	Data traffic	**`I_SERIAL_DUPLEX_HALF`** **`I_SERIAL_DUPLEX_FULL`**
`I_SERIAL_FLOW_CTRL`	Flow control	**`I_SERIAL_FLOW_NONE`** **`I_SERIAL_FLOW_XON`** **`I_SERIAL_FLOW_RTS_CTS`** **`I_SERIAL_FLOW_DTR_DSR`**
`I_SERIAL_READ_EOI`	EOI indicator for reads	**`I_SERIAL_EOI_NONE`** **`I_SERIAL_EOI_BIT8`** **`I_SERIAL_EOI_CHAR`** | (*n*)
`I_SERIAL_WRITE_EOI`	EOI indicator for writes	**`I_SERIAL_EOI_NONE`** **`I_SERIAL_EOI_BIT8`**
`I_SERIAL_RESET`	Interface state	(none)

Function Name	Action
`iserialstat`	Gets the following information about the RS-232 interface:

Request	Characteristic	Value
I_SERIAL_BAUD	Data rate	2400, 9600, etc.
I_SERIAL_PARITY	Parity	**I_SERIAL_PAR_***
I_SERIAL_STOP	Stop bits / frame	**I_SERIAL_STOP_***
I_SERIAL_WIDTH	Data bits / frame	**I_SERIAL_CHAR_***
I_SERIAL_DUPLEX	Data traffic	**I_SERIAL_DUPLEX_***
I_SERIAL_MSL	Modem status lines	**I_SERIAL_DCD** **I_SERIAL_DSR** **I_SERIAL_CTS** **I_SERIAL_RI** **I_SERIAL_TERI** **I_SERIAL_D_DCD** **I_SERIAL_D_DSR** **I_SERIAL_D_CTS**
I_SERIAL_STAT	Misc. status	**I_SERIAL_DAV** **I_SERIAL_TEMT** **I_SERIAL_PARITY** **I_SERIAL_OVERFLOW** **I_SERIAL_FRAMING** **I_SERIAL_BREAK**
I_SERIAL_READ_BUFSZ	Receive buffer size	Number of bytes
I_SERIAL_READ_DAV	Data available	Number of bytes
I_SERIAL_FLOW_CTRL	Flow control	**I_SERIAL_FLOW_***
I_SERIAL_READ_EOI	EOI indicator for reads	**I_SERIAL_EOI***
I_SERIAL_WRITE_EOI	EOI indicator for writes	**I_SERIAL_EOI***

Function Name	Action
`iserialmclctrl`	Sets or Clears the modem control lines. Modem control lines are either **`I_SERIAL_RTS`** or **`I_SERIAL_DTR`**.
`iserialmclstat`	Gets the current state of the modem control lines.
`iserialbreak`	Sends a break to the instrument. Break time is 10 character times, with a minimum time of 50 milliseconds and a maximum time of 250 milliseconds.

Using RS-232 Device Sessions

An RS-232 **device session** allows direct access to a device, regardless of the type of interface to which the device is connected. The specifics of the interface are hidden from the user.

Addressing RS-232 Devices

To create a device session, specify the interface logical unit or symbolic name, followed by a device logical address of **488**. The device address of **488** tells SICL that communication is with a device that uses the IEEE 488.2 standard command structure. For other interfaces (such as GPIB), SICL supports the concept of primary and secondary addresses. However, for RS-232, the only primary address supported is **488**. SICL does not support secondary addressing on RS-232 interfaces.

NOTE

If a device does not "speak" IEEE 488.2, you can still use SICL to communicate with the device. However, some SICL functions that work only with device sessions may not operate correctly. See "SICL Function Support for RS-232 Device Sessions" for details.

The interface logical unit and symbolic name are defined by running the **IO Config** utility. To open **IO Config**, click the **Agilent IO Libraries Control** and then click **Run IO Config.** See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on **IO Config**. Some example addresses for RS-232 device sessions follow.

```
COM1,488
serial,488
```

Examples of opening a device session with an RS-232 device follow.

C example:

```
INST dmm;
dmm = iopen ("com1,488");
```

Visual Basic example:

```
Dim dmm As Integer
dmm = iopen ("com1,488"
```

SICL Function Support for RS-232 Device Sessions

This section describes how some SICL functions are implemented for RS-232 device sessions. There are specific device session interrupts that can be used. See **`isetintr`** in *Chapter 11 - SICL Language Reference* for information on RS-232 device session interrupts.

`iprintf`, `iscanf`, `ipromptf`	SICL's formatted I/O routines depend on the concept of an EOI indicator. Since RS-232 does not define an EOI indicator, SICL uses the newline character (**`\n`**) by default. You cannot change this with a device session. However, you can use the **`iserialctrl`** function with an interface session. See " SICL Function Support for RS-232 Interface Sessions" in this chapter for details.
`ireadstb`	Sends the IEEE 488.2 command **`*STB?`** to the instrument, followed by the newline character (**`\n`**). It then reads the ASCII response string and converts it to an 8-bit integer. This will work only if the instrument understands this command.
`itrigger`	Sends the IEEE 488.2 command **`*TRG`** to the instrument, followed by the newline character (**`\n`**). This will work only if the instrument understands this command.
`iclear`	Sends a break, aborts any pending writes, discards any data in the receive buffer, resets any flow control states (such as XON/XOFF), and resets any error conditions. To reset the interface without sending a break, use: **`iserialctrl`** (*id*, `I_SERIAL_RESET`, 0)
`ionsrq`	Installs a service request handler for this session. Service requests are supported for both device sessions and interface sessions. See "SICL Function Support for RS-232 Interface Sessions" in this chapter for details.

Example: RS-232 Device Session (C)

This example program takes a measurement from a DVM using a SICL device session.

> **NOTE**
>
> This example program was tested with a 34401A Digital Voltmeter. When you run the program with a serial connection to the 34401A, be sure that DTR/DSR flow control is set for the serial port. Otherwise, the program will appear not to work.

```
/* ser_dev.c
   This example program takes a measurement from a DVM
   using a SICL device session.
*/
#include <sicl.h>
#include <stdio.h>
#include <stdlib.h>

#if !defined(WIN32)
   #define LOADDS __loadds
#else
   #define LOADDS
#endif

void SICLCALLBACK LOADDS error_handler (INST id, int
error) {

   printf (“Error: %s\n”, igeterrstr (error));
   exit (1);
}

main()
{
   INST dvm;
   double res;

   #if defined(__BORLANDC__) && !defined(__WIN32__
      _InitEasyWin(); /* required for Borland EasyWin
                         programs */
   #endif
```

```
  /* Log message and terminate on error */
  ionerror (error_handler);

  /* Open the multimeter session */
  dvm = iopen (“COM1,488”);
  itimeout (dvm, 10000);

  /* Prepare the multimeter for measurements */
  iprintf (dvm,”*RST\n”);
  iprintf (dvm,”SYST:REM\n”);

  /* Take a measurement */
  iprintf (dvm,”MEAS:VOLT:DC?\n”);

  /* Read the results */
  iscanf (dvm,”%lf”,&res);

  /* Print the results */
  printf (“Result is %f\n”,res);

  /* Close the voltmeter session */
  iclose (dvm);

  /* This call is a no-op for WIN32 programs */
  _siclcleanup();

  return 0;
}
```

Example: RS-232 Device Session (Visual Basic)

This example program takes a measurement from a DVM using a SICL device session.

NOTE

This example program was tested with a 34401A Digital Voltmeter. When you run the program with a serial connection to the 34401A, be sure that DTR/DSR flow control is set for the serial port. Otherwise, the program will appear not to work.

```
‘  ser_dev.bas
‘  This example program takes a measurement from a DVM
‘  using a SICL device session.

Sub Main ()
   Dim dvm As Integer
   Dim res As Double
   Dim argcount As Integer

   ‘ Open the multimeter session
   dvm = iopen(“COM1,488”)
   Call itimeout(dvm, 10000)

   ‘ Prepare the multimeter for measurements
   argcount = ivprintf(dvm, “*RST” + Chr$(10), 0&)

   argcount = ivprintf(dvm, “SYST:REM” + Chr$(10), 0&)

   ‘ Take a measurement
   argcount = ivprintf(dvm, “MEAS:VOLT:DC?” + Chr$(10))

   ‘ Read the results
   argcount = ivscanf(dvm, “%lf”, res)

   ‘ Print the results
   MsgBox “Result is “ + Format(res), MB_ICON_EXCLAMATION

   ‘ Close the multimeter session
   Call iclose(dvm)

‘  Tell SICL to clean up for this task
   Call siclcleanup

End Sub
```

Using RS-232 Interface Sessions

RS-232 **interface sessions** can be used to get or set the characteristics of the RS-232 interface. Examples of some of these characteristics are baud rate, parity, and flow control. There are specific interface session interrupts that can be used. See **`isetintr`** in *Chapter 11 - SICL Language Reference* for information on RS-232 interface session interrupts.

Addressing RS-232 Interfaces

To create an **interface session** on RS-232, specify the interface logical unit or symbolic name in the *addr* parameter of the **`iopen`** function. The interface logical unit and symbolic name are defined by running the **`IO Config`** utility. To open **`IO Config`**, click the **`Agilent IO Libraries Control`** and then click **`Run IO Config.`** See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on **`IO Config`**. Some example addresses for RS-232 interface sessions follow.

`COM1`	An interface symbolic name
`serial`	An interface symbolic name
`1`	An interface logical unit

These examples open an interface session with the RS-232 interface.

C example:

```
INST intf;
intf = iopen (“COM1”);
```

Visual Basic example:

```
Dim intf As Integer
intf = iopen (“COM1”)
```

SICL Function Support for RS-232 Interface Sessions

This section describes how some SICL functions are implemented for RS-232 interface sessions.

iwrite, **iread**	All I/O functions (non-formatted and formatted) work the same as for device sessions. However, it is recommended that all I/O be performed with device sessions to make your programs easier to maintain.
ixtrig	Provides a method of triggering using either the DTR or RTS modem status line. This function clears the specified modem status line, waits 10 milliseconds, then sets it again. Specifying **I_TRIG_STD** is the same as specifying **I_TRIG_SERIAL_DTR**.
itrigger	Pulses the DTR modem control line for 10 milliseconds.
iclear	Sends a break, aborts any pending writes, discards any data in the receive buffer, resets any flow control states (such as **XON/XOFF**), and resets any error conditions. To reset the interface without sending a break, use: **iserialctrl** (*id*, I_SERIAL_RESET, 0)
ionsrq	Installs a service request handler for this session. The concept of service request (SRQ) originates from GPIB. On a GPIB interface, a device can request service from the controller by asserting a line on the interface bus. RS-232 does not have a specific line assigned as a service request line. However, you can assign one of the modem status lines (RI, DCD, CTS, or DSR) as the service request line by running the **IO Config** utility. Any transition on the designated service request line will cause an SRQ handler in your program to be called. (Be sure not to set the SRQ line to CTS or DSR if you are also using that line for hardware flow control.) Service requests are supported for both device sessions and interface sessions. When the designated SRQ line changes state, the RS-232 driver calls all SRQ handlers installed by either device sessions or interface sessions.

iserialctrl	Sets the characteristics of the serial interface. The following requests are clarified: ■ **I_SERIAL_DUPLEX:** The duplex setting determines whether data can be sent and received simultaneously. Setting full duplex allows simultaneous send and receive data traffic. Setting half duplex (the default) will cause reads and writes to be interleaved, so that data is flowing in only one direction at any given time. (The exception to this is if **XON/XOFF** flow control is used.) ■ **I_SERIAL_READ_BUFSZ**: The default read buffer size is 2048 bytes. ■ **I_SERIAL_RESET**: Performs the same function as the **iclear** function on an interface session, except that a break is not sent.
iserialstat	Gets the characteristics of the serial interface. The following requests are clarified: ■ **I_SERIAL_MSL**: Gets the state of the modem status line. Because of the way Windows supports RS-232, the **I_SERIAL_RI** bit will never be set. However, the **I_SERIAL_TERI** bit will be set when the RI modem status line changes from high to low. ■ **I_SERIAL_STAT**: Gets the status of the transmit and receive buffers and the errors that have occurred since the last time this request was made. Only the error bits (**I_SERIAL_PARITY, I_SERIAL_OVERFLOW, I_SERIAL_FRAMING,** and **I_SERIAL_BREAK**) are cleared. The **I_SERIAL_READ_DAV** and **I_SERIAL_TEMT** bits reflect the status of the buffers at all times. ■ **I_SERIAL_READ_DAV**: Gets the current amount of data available for reading. This shows how much data is in Windows' receive buffer, not how much data is in the buffer used by the formatted input functions such as **iscanf**.

iserial-mclctrl	Controls the modem control lines RTS and DTR. If one of these lines is being used for flow control, you cannot set that line with this function.
iserial-mclstat	Determines the current state of the modem control lines. If one of these lines is being used for flow control, this function may not give the correct state of that line.

Example: RS-232 Interface Session (C)

```
/*ser_intf.c
   This program gets the current configuration of the
   serial port, sets it to 9600 baud, no parity, 8 data
bits, and 1 stop bit, and prints the old configuration.
*/
#include <stdio.h>
#include <sicl.h>

main()
{
   INST intf;                  /* interface session id */
   unsigned long baudrate, parity, databits, stopbits;
   char *parity_str;

   #if defined(__BORLANDC__) && !defined(__WIN32__)
   _InitEasyWin(); /* reqd for Borland EasyWin programs */
   #endif

   /* Log message and exit program on error */
   ionerror (I_ERROR_EXIT);

   /* open RS-232 interface session */
   intf = iopen ("COM1");
   itimeout (intf, 10000);

   /* get baud rate, parity, data bits, and stop bits */
   iserialstat (intf, I_SERIAL_BAUD,   &baudrate);
   iserialstat (intf, I_SERIAL_PARITY, &parity);
   iserialstat (intf, I_SERIAL_WIDTH,  &databits);
   iserialstat (intf, I_SERIAL_STOP,   &stopbits);
```

```
  /* determine string to display for parity */
  if   (parity == I_SERIAL_PAR_NONE) parity_str = “NONE”;
  else if (parity == I_SERIAL_PAR_ODD) parity_str =
“ODD”;
  else if (parity == I_SERIAL_PAR_EVEN) parity_str =
“EVEN”;
  else if (parity == I_SERIAL_PAR_MARK)   parity_str =
“MARK”;
  else  /*parity == I_SERIAL_PAR_SPACE*/ parity_str =
“SPACE”;

  /* set to 9600,NONE,8,1 */
  iserialctrl (intf, I_SERIAL_BAUD,   9600);
  iserialctrl (intf, I_SERIAL_PARITY,
I_SERIAL_PAR_NONE);
  iserialctrl (intf, I_SERIAL_WIDTH,  I_SERIAL_CHAR_8);
  iserialctrl (intf, I_SERIAL_STOP,   I_SERIAL_STOP_1);

   /* Display previous settings */
  printf(“Old settings:  %5ld,%s,%ld,%ld\n”,
         baudrate, parity_str, databits, stopbits);

  /* close port */
  iclose (intf);

  /* This call is a no-op for WIN32 programs. */
  _siclcleanup();

  return 0;
}
```

Example: RS-232 Interface Session (Visual Basic)

```
‘  ser_intf.bas
‘  This program gets the current configuration of the
‘   serial port, sets it to 9600 baud, no parity, 8 data
‘   bits, and 1 stop bit and prints the old configuration

Sub main ()
  Dim intf As Integer
  Dim baudrate As Long
  Dim parity As Long
  Dim databits As Long
  Dim stopbits As Long
```

```
  Dim parity_str As String
  Dim msg_str As String

  ‘ open RS-232 interface session
  intf = iopen(“COM1”)
  Call itimeout(intf, 10000)

  ‘ get baud rate, parity, data bits, and stop bits
  Call iserialstat(intf, I_SERIAL_BAUD, baudrate)
  Call iserialstat(intf, I_SERIAL_PARITY, parity)
  Call iserialstat(intf, I_SERIAL_WIDTH, databits)
  Call iserialstat(intf, I_SERIAL_STOP, stopbits)

  ‘ determine string to display for parity
  Select Case parity
  Case I_SERIAL_PAR_NONE
    parity_str = “NONE”
  Case I_SERIAL_PAR_ODD
    parity_str = “ODD”
  Case I_SERIAL_PAR_EVEN
    parity_str = “EVEN”
  Case I_SERIAL_PAR_MARK
    parity_str = “MARK”
  Case Else
    parity_str = “SPACE”
  End Select

  ‘ set to 9600,NONE,8, 1
  Call iserialctrl(intf, I_SERIAL_BAUD, 9600)
  Call iserialctrl(intf, I_SERIAL_PARITY, I_SERIAL_PAR_NONE)
  Call iserialctrl(intf, I_SERIAL_WIDTH, I_SERIAL_CHAR_8)
  Call iserialctrl(intf, I_SERIAL_STOP, I_SERIAL_STOP_1)

  ‘ display previous settings
  msg_str = “Old settings: “ + Str$(baudrate) + “,” +
       parity_str + “,” + Str$(databits) + “,” +
       Str$(stopbits)
  MsgBox msg_str, MB_ICON_EXCLAMATION

  ‘ close port
  Call iclose(intf)
  ‘  Tell SICL to clean up for this task
  Call siclcleanup

End Sub
```

Notes:

8

Using SICL with LAN

Using SICL with LAN

This chapter shows how to open a communications session and communicate with devices over a Local Area Network (LAN). The example programs in this chapter are also provided in **C\SAMPLES\MISC** (for C/C++) and **VB\SAMPLES\MISC** (for Visual Basic). The chapter includes:

- LAN Overview
- Using LAN_gatewayed Sessions
- Using LAN Interface Sessions
- Using Locks and Threads over LAN
- Using Timeouts with LAN

LAN Overview

A LAN extends control of instrumentation beyond the limits of typical instrument interfaces. LAN is only supported with 32-bit SICL on Windows 95, Windows 98, Windows 2000, and Windows NT. LAN is only supported with 32-bit Visual Basic version 4.0 and above. Also, the GPIO interface is **not** supported with SICL over LAN.

The LAN software provided with SICL allows instrumentation control over a LAN. By using standard LAN connections, instrument control can be driven from a computer that does not have a special interface for instrument control. To start or stop the LAN server on a Windows 95, Windows 98, Windows 2000, or Windows NT system, see the *Agilent IO Libraries Installation and Configuration Guide for Windows.*

LAN Client/Server Model

The LAN software provided with SICL uses the client/server model of computing. **Client/server computing** refers to a model where an application, the **client**, does not perform all the necessary tasks of the application itself. Instead, the client makes requests of another computing device, the **server**, for certain services. Examples that you may have in your workplace include shared file servers, print servers, or database servers.

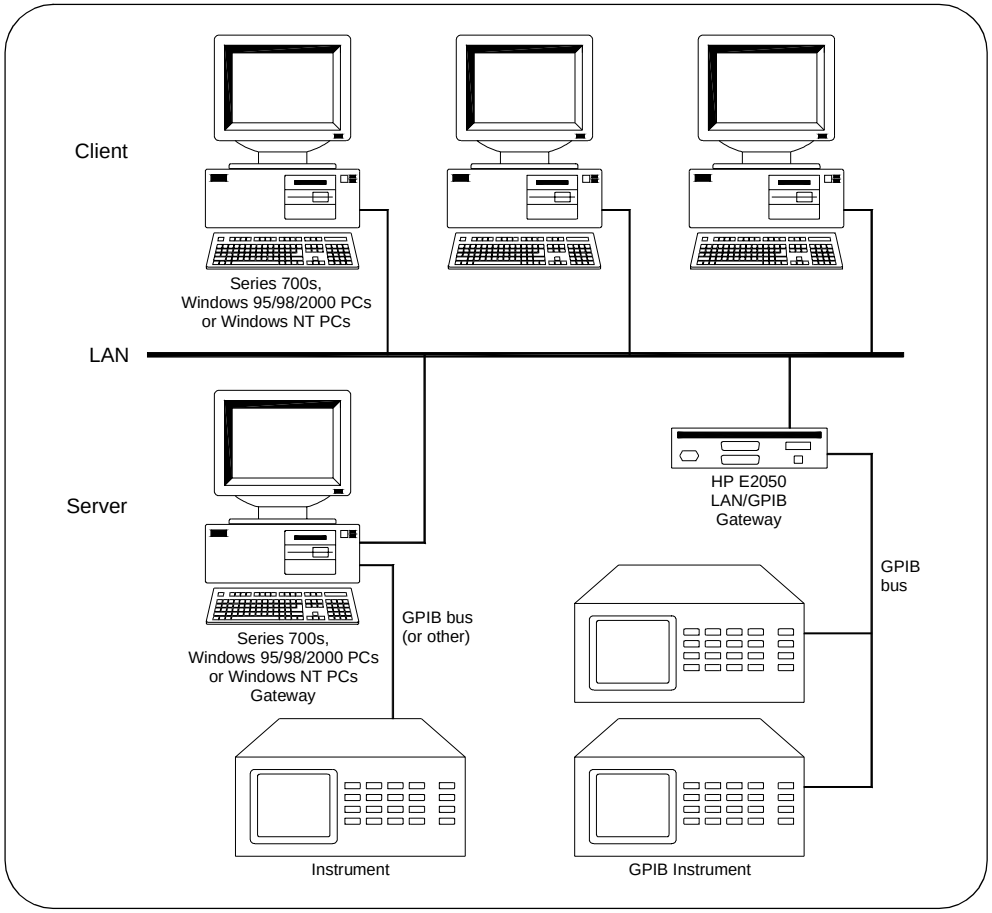
The use of LAN for instrument control also provides other advantages associated with client/server computing, such as resource sharing by multiple applications/people within an organization or distributed control, where the computer running the application controlling the devices need not be in the same room (or even the same building) as the devices.

LAN Hardware Architecture

As shown in the following figure, a LAN client computer system (a Series 700 HP-UX workstation, a Windows 95/98/2000 PC, or a Windows NT PC) makes SICL requests over the network to a LAN server (a Series 700 HP-UX workstation, a Windows 95/98/2000 PC, a Windows NT PC, or an E2050 LAN/GPIB Gateway).

The LAN server is connected to the instrumentation or devices that must be controlled. Once the LAN server has completed the requested operation on the instrument or device, the LAN server sends a reply to the LAN client. This reply contains any requested data and status information that indicates whether the operation was successful.

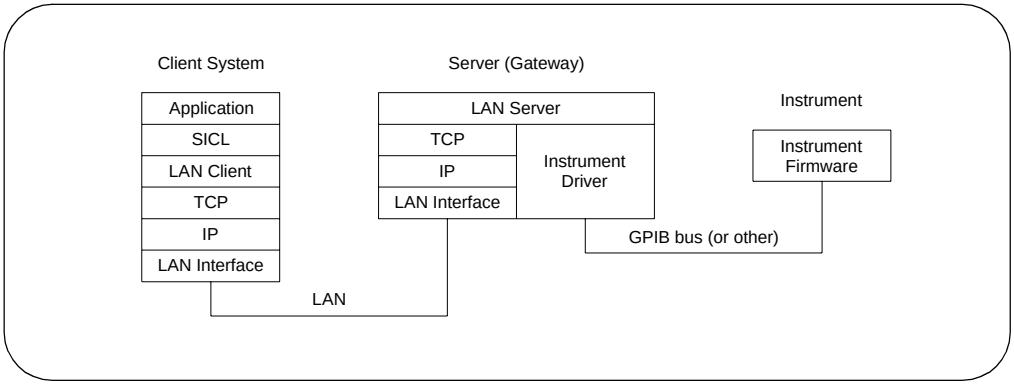
The LAN server acts as a **gateway** between the LAN that the client system supports, and the instrument-specific interface that the device supports. Due to the LAN server's gateway functionality, we refer to devices or interfaces that are accessed via one of these LAN-to-instrument_interface gateways as being a LAN-gatewayed device or a LAN-gatewayed interface.



Using the LAN Client and LAN Server (Gateway)

LAN Software Architecture

As shown in the following figure, the client system contains the LAN client software and the LAN software (TCP/IP) needed to access the server (gateway). The gateway contains the LAN server software, LAN (TCP/IP) software, and the instrument driver software needed to communicate with the client and to control the instruments or devices connected to the gateway..



LAN Software Architecture

LAN Networking Protocols

The LAN software provided with SICL is built on top of standard LAN networking protocols. There are two LAN networking protocols provided with the SICL software. You can use one or both of these protocols when configuring your systems (via the **`IO Config`** utility) to use SICL over LAN.

- **SICL LAN Protocol** is a networking protocol developed by Hewlett-Packard that is compatible with all existing SICL LAN products. This LAN networking protocol is the default choice in the **`IO Config`** utility when configuring LAN for SICL. The SICL LAN Protocol on Windows 95/98/2000 and Windows NT supports SICL operations over the LAN to GPIB and RS-232 interfaces.

- **TCP/IP Instrument Protocol** is a networking protocol developed by the VXIbus Consortium based on the SICL LAN Protocol which permits interoperability of LAN software from different vendors that meet the VXIbus Consortium standards.

This LAN networking protocol may not be implemented with all SICL LAN products at this time. The TCP/IP Instrument Protocol on Windows 95, Windows 98, Windows 2000, and Windows NT supports SICL operations over the LAN to GPIB interface. Also, some SICL operations are not supported when using the TCP/IP Instrument Protocol.

When using either of these networking protocols, the LAN software provided with SICL uses the TCP/IP protocol suite to pass messages between the LAN client and the LAN server. The server accepts device I/O requests over the network from the client and then proceeds to execute those I/O requests on a local interface, such as GPIB.

You can use both LAN networking protocols with a LAN client by configuring *both* the SICL LAN Protocol and the TCP/IP Instrument Protocol on the LAN client system using the **`IO Config`** utility. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on running **`IO Config`**.

Then, use the name of the interface supporting the protocol you want to use in each SICL **`iopen`** call of your program. See “Communicating with LAN Devices” in this chapter for details on creating communications sessions with SICL over LAN using each of these protocols. The LAN server does *not* support simultaneous connections from LAN clients using the SICL LAN Protocol and from other LAN clients using the TCP/IP Instrument Protocol.

LAN Clients and Threads

You can use multi-threaded designs (where SICL calls are made from multiple threads) in WIN32 SICL applications over LAN. However, only one thread is permitted to access the LAN driver at a time. This sequential handling of individual threads by the LAN driver prevents multiple threads from colliding or overwriting one another. Requests are handled sequentially even if they are intended for different LAN servers.

Use multiple processes to process concurrent threads simultaneously with SICL over LAN. For more information on using threads in SICL applications, see *Chapter 3 - Programming with SICL*. Also, see “Using Locks and Threads over LAN” in this chapter for information on using locks in multi-threaded applications.

LAN Servers

SICL includes software required to allow a Windows 95, Windows 98, Wikndows 2000, or Windows NT PC to act as a LAN-to-instrument_interface gateway. To use this capability, the PC must have a local interface configured for I/O. The supported interfaces for this release are GPIB and RS-232 with the SICL LAN Protocol, and GPIB with the TCP/IP Instrument Protocol.

The LAN server does *not* support VXI operations with either protocol. Timing of operations performed remotely over a network will be different from timing of operations performed locally. The extent of the timing difference will, in part, depend on the bandwidth of, and the traffic on, the network being used.

SICL LAN Configuration and Performance

As with other client/server applications on a LAN, when deploying an application that uses SICL over LAN, consideration must be given to the performance and configuration of the network to which the client and server will be attached. If the network to be used is not a dedicated LAN or otherwise isolated via a bridge or other network device, current use of the LAN must be considered.

Depending on the amount of data to be transferred over the LAN via the SICL application, performance problems could be experienced by the SICL application or other network users if sufficient bandwidth is not available. This is not unique to SICL over LAN, but is a general design consideration when deploying any client/server application.

If you have questions concerning the ability of your network to handle SICL traffic, consult with your network administrator or network equipment providers.

SICL LAN Functions

Function Name	Action
`ilantimeout`	Sets LAN timeout value
`ilangettimeout`	Returns LAN timeout value
`igetgatewaytype`	Indicates whether the session is via a LAN gateway

Using LAN-gatewayed Sessions

Communicating with a device over LAN vai a LAN-to-instrument_interface gateway preserves the functionality of the gatewayed-interface with a few exceptions. (See “SICL Function Support with LAN-gatewayed Sessions” in this chapter.) Thus, for most operations over an interface (such as GPIB connected directly to your controller), can also be performed over a remote interface via the LAN gateway.

The only portions of your application that must be changed are the addresses passed to the **`iopen`** calls (unless those addresses are stored in a configuration file, in which case no changes to the application itself are required). The address used for a local interface must have a LAN prefix added so the SICL software knows to direct the request to a LAN server on the network.

Addressing Devices/Interfaces with LAN-gatewayed Sessions

To create a LAN-gatewayed session, specify the LAN’s interface logical unit or interface name, the IP address or hostname of the server machine, and the address of the remote interface or device in the *addr* parameter of the **`iopen`** function. The interface logical unit and interface name are defined by running the **`IO Config`** utility.

To open the the **`IO Config`** utility, click the **`Agilent IO Libraries Control`** and then click **`Run IO Config`**. See the Agilent *IO Libraries Installation and Configuration Guide for Windows* for information on running **`IO Config`**. Some examples of LAN-gatewayed addresses follow.

NOTE

If you are using the IP address of the server machine rather than the hostname, you must use the bracket (not the comma) notation.

`lan,128.10.0.3:gpib` (Incorrect)
`lan[128.10.0.3]:gpib` (Correct)

lan[instserv]:GPIB,7	A device address corresponding to the device at primary address **7** on the **GPIB** interface attached to the machine named **instserv**.
lan[instserv.hp.com]:gpib,7	A device address corresponding to the device at primary address **7** on the **gpib** interface attached to the machine named **instserv** in the **hp.com** domain. (Fully qualified domain names may be used.)
lan1[128.10.0.3]:GPIB0,3,2	A device address corresponding to the device at primary address **3**, secondary address **2**, on the **GPIB0** interface attached to the machine with IP address **128.10.0.3**
lan1[intserv]:GPIB2	An interface address corresponding to the **GPIB2** interface attached to the machine named **intserv**.
30,intserv:gpib,3,2	A device address corresponding to the device at primary address **3**, secondary address **2**, on the **gpib** interface attached to the machine named **intserv**. (30 is the default logical unit for LAN.)
lan[intserv]:GPIB,cmdr	A commander session with the **GPIB** interface attached to the machine named **intserv.** (Assuming the server supports GPIB commander sessions.)

This table shows the relationship between the address passed to iopen, the session type returned by **igetsesstype**, the interface type returned by **igetintftype**, and the value returned by **igetgatewaytype**.

Address	Session Type	Interface Type	Gateway Type
`lan`	`I_SESS_INTF`	`I_INTF_LAN`	`I_INTF_NONE`
`lan[instserv]:hpib`	`I_SESS_INTF`	`I_INTF_GPIB`	`I_INTF_LAN`
`lan[instserv]:hpib,7`	`I_SESS_DEV`	`I_INTF_GPIB`	`I_INTF_LAN`
`hpib`	`I_SESS_INTF`	`I_INTF_GPIB`	`I_INTF_NONE`
`hpib,7`	`I_SESS_DEV`	`I_INTF_GPIB`	`I_INTF_NONE`

SICL Function Support with LAN-gatewayed Sessions

A gatewayed-session to a remote interface provides the same SICL function support as if the interface was local, with the following exceptions or qualifications.

Type of Functions	SICL Functions NOT Supported
SICL functions *not* supported over LAN using either protocol	**`iblockcopy, imap, imapinfo, ipeek, ipoke, ipopfifo, ipushfifo, iunmap`**
SICL functions, *in addition to those listed above*, *not* supported with the TCP/IP Instrument Protocol	All RS-232/serial specific functions **`igetlu, ionintr, isetintr, igetintfsess, igetonintr, igpibgett1delay, igpibppoll igpibppollconfig, igpibppollresp, igpibsett1delay`**

For the **igetdevaddr**, **igetintftype**, and **igetsesstype** functions to be supported with the TCP/IP Instrument Protocol, the remote address strings *must* follow the TCP/IP Instrument Protocol naming conventions – `gpib0`, `gpib1`, etc. For example:

```
gpib0,7
gpib1,7,2
gpib2
```

However, since the interface names at the remote server may be configurable, this is not guaranteed. Correct behavior of **iremote** and **iclear** depend on the correct address strings being used. When **iremote** is executed over the TCP/IP Instrument Protocol, **iremote** also sends the LLO (local lockout) message in addition to placing the device in the remote state.

Any of the following functions may timeout over LAN, even those functions that cannot timeout over local interfaces. (See “Using Timeouts with LAN” in this chapter for more details.) These functions all cause a request to be sent to the server for execution.

All GPIB specific functions
All RS-232/serial specific functions
iabort, iclear, iclose, iflush, ifread, ifwrite, igetintfsess, ilocal, ilock, ionintr, ionsrq, iopen, iprintf, ipromptf, iread, ireadstb, iremote, iscanf, isetbuf, isetintr, isetstb, isetubuf, itrigger, iunlock, iversion, iwrite, ixtrig

These SICL functions perform as follows with LAN-gatewayed sessions.

idrvrversion	Returns the version numbers from the server.
iwrite, iread	**actualcnt** may be reported as 0 when some bytes were transferred to or from the device by the server. This can happen if the client times out while the server is in the middle of an I/O operation.

Example: LAN-gatewayed Session (C)

This example program opens a GPIB device session via a LAN-to-GPIB gateway. This example is the same as the example in *Chapter 4 - Usisng SICL with GPIB*, execpt the addresses passed to the **iopen** calls are modified. The addresses in this example assume a machine with hostname **instserv** is acting as a LAN-to-GPIB gateway.

```
/* landev.c
This example program sends a scan list to a switch and
while looping closes channels and takes measurements.*/
#include <sicl.h>
#include <stdio.h>
```

```
main() {
   INST dvm;
   INST sw;

   double res;
   int i;
   /* Print message and terminate on error */
   ionerror (I_ERROR_EXIT);

   /* Open the multimeter and switch sessions */
   dvm = iopen ("lan[instserv]:hpib,9,3");
   sw = iopen ("lan[instserv]:hpib,9,14");
   itimeout (dvm, 10000);
   itimeout (sw, 10000);

   /*Set up trigger*/
   iprintf (sw, "TRIG:SOUR BUS\n");

   /*Set up scan list*/
   iprintf (sw,"SCAN (@100:103)\n");
   iprintf (sw,"INIT\n");

   for (i=1;i<=4;i++)   {
      /* Take a measurement */
      iprintf (dvm,"MEAS:VOLT:DC?\n");

      /* Read the results */
      iscanf (dvm,"%lf", &res);

      /* Print the results */
      printf ("Result is %f\n",res);
      /*Trigger to close channel*/
      iprintf (sw, "TRIG\n");
   }
   /* Close the multimeter and switch sessions */
   iclose (dvm);
   iclose (sw);
}
```

Example: LAN-gatewayed Session (Visual Basic)

This example program opens a GPIB device session via a LAN-to-GPIB gateway. This example is the same as the example in *Chapter 4 - Usisng SICL with GPIB*, execpt the addresses passed to the **iopen** calls are modified. The addresses in this example assume a machine with hostname **instserv** is acting as a LAN-to-GPIB gateway.

```
‘ landev.bas
‘ This program sends a scan list to a switch and while
‘ looping closes channels and takes measurements.

Attribute VB_Name = “Module1”

Public Sub lanmain()
   Dim dvm As Integer, sw As Integer
   Dim nargs As Integer, I As Integer
   Dim res As Double
   Dim actual As Long
   Dim res1 As String

   ‘  Set up an error handler within this subroutine that
   ‘  will get called if a SICL error occurs.
   On Error GoTo ErrorHandler

   ‘Open the multimeter and switch sessions
   dvm = iopen(“lan[intserv]:hpib,9,3”)
   sw = iopen(“lan[intserv]:hpib,9,14”)

   Call itimeout(dvm, 10000)
   Call itimeout(sw, 10000)

   ‘set up the trigger
   nargs = iwrite(id, “TRIG:SOUR BUS” + Chr$(10) + Chr$(0), 14, 1, actual)

   ‘set up scan list
   nargs = iwrite(id, “SCAN (@100:103)” + Chr$(10) + Chr$(0), 15, 1, actual)
   nargs = iwrite(id, “INIT” + Chr$(10) + Chr$(0), 5, 1, actual)

For I = 1 To 4 Step 1
   nargs = iwrite(id, “MEAS:VOLT:DC?” + Chr$(10) + Chr$(0), 14, 1, actual)
   nargs = iread(id, res1, l, &H0&, actual)
```

```
   MsgBox “Result is”
   MsgBox res1

   nargs = iwrite(id, “TRIG” + Chr$(10) + Chr$(0), 5, 1, actual)
   Next I

   Dim x As Integer
   x = iclose(dvm)
   x = iclose(sw)

   Exit Sub

ErrorHandler:

‘  Display the error message in the txtResponse TextBox.
   txtResponse.Text = “*** Error : “ + Error$
   MsgBox txtResponse.Text
‘  Close the device session if iopen was successful.
   If id <> 0 Then
     iclose (id)
   End If

   Exit Sub
End Sub
```

Using LAN Interface Sessions

The LAN interface, unlike most other supported SICL interfaces, does not allow for direct communication with devices via interface commands. LAN interface sessions, if used at all, will typically be used only for setting the client side LAN timeout. (See “Using Timeouts with LAN” in this chapter.)

Addressing LAN Interface Sessions

To create a LAN interface session, specify the interface logical unit or interface name in the *addr* parameter of the **`iopen`** function. The interface logical unit and interface name are defined by running the **`IO Config`** utility.

To open the the **`IO Config`** utility, click the **`Agilent IO Libraries Control`** and then click **`Run IO Config`**. See the Agilent *IO Libraries Installation and Configuration Guide for Windows* for information on running **`IO Config`**. Some examples of LAN interface addresses follow.

`lan`	A LAN interface address using the interface name **`lan`**.
`30`	A LAN interface address using the logical unit 30. (**`30`** is the default logical unit for LAN.)

SICL Function Support with LAN Interface Sessions

These SICL functions are *not* supported over LAN interface sessions and return **`I_ERR_NOTSUPP`**.

All GPIB specific functions
All serial specific functions
All formatted I/O routines
`iwrite, iread, ilock, iunlock, isetintr, itrigger, ixtrig, ireadstb, isetstb, imapinfo, ilocal, iremote`

These SICL functions perform as follows with LAN interface sessions.

iclear	Performs no operation, returns **I_ERR_NOERROR.**
ionsrq	Performs no operation against LAN gateways for SICL, returns **I_ERR_NOERROR**.
ionintr	Performs no operation, returns **I_ERR_NOERROR**.
igetluinfo	Returns information about local interfaces only. Does not return information about remote interfaces that are being accessed via a LAN-to-instrument_interface gateway.

Using Locks and Threads over LAN

If two or more threads are accessing the same device or interface using two or more different sessions over LAN, and are using SICL locks to synchronize access, some scenarios may cause timeouts or may “hang” an application that does not use timeouts. For proper operation, all threads that use their own sessions to access the same device or interface should use the same string to identify the device or interface in their calls to **`iopen`**. Therefore, the following scenarios **should be avoided**.

- Using a hostname to identify the remote host in one call to **`iopen`** while using an alias or IP address to identify the same host in another call to **`iopen`**.
- Using a device symbolic name in one call to **`iopen`** (such as ”**`dmm`**”, where “**`dmm`**” equals “**`gpib,1`**”) while using the fully specified device name (such as “**`gpib,1`**”) in another call.
- Using a remote interface’s logical unit (such as “**`7`**”) in one call while using the remote interface’s symbolic name (such as “**`gpib`**”) in another.
- Using **`igetintfsess`** to open an interface session (which internally uses the logical unit to identify the remote interface) while opening the interface with its symbolic name for another session.

You can avoid each scenario by always using the same strings to identify the same device or interface in multi-threaded applications. You can also use the **`igetintfsess`** function if other sessions use the logical unit to specify the interface instead of the interface’s symbolic name.

If any thread uses **`ilock`** and **`iunlock`** to synchronize access to a particular device or interface, all threads accessing that same device or interface using a different session must also use **`ilock`** and **`iunlock`**. WIN32 synchronization techniques may also be used to ensure that a thread does not attempt I/O (**`iread/iwrite`**, etc.) to a device already locked via a different session from a different thread within the same process.

If a session has an interface locked, and if a different thread using its own session attempts to lock a device on that interface, the device lock will be held off either until the interface is unlocked by the other thread, or until a timeout occurs on the device lock. This is different from how **`ilock`** works on other interfaces (where a lock on a device when the device's interface is already locked will not hold off the **`ilock`** operation, but rather will hold off any subsequent I/O to the device).

Using Timeouts with LAN

The client/server architecture of the LAN software requires use of two timeout values: one for the client and one for the server. The server's timeout value is the SICL timeout value specified with the **`itimeout`** function. The client's timeout value is the LAN timeout value, which may be specified with the **`ilantimeout`** function.

When the client sends an I/O request to the server, the timeout value specified with **`itimeout`** or with the SICL default is passed with the request. The server uses that timeout in performing the I/O operation, just as if that timeout value had been used on a local I/O operation.

If the server's operation is not completed in the specified time, the server sends a reply to the client that indicates that a timeout occurred, and the SICL call made by the application returns **`I_ERR_TIMEOUT`**.

When the client sends an I/O request to the server, it starts a timer and waits for the reply from the server. If the server does not reply in the time specified, the client stops waiting for the reply from the server and returns **`I_ERR_TIMEOUT`** to the application.

LAN Timeout Functions

The **`ilantimeout`** and **`ilangettimeout`** functions can be used to set or query the current LAN timeout value. They work much like the **`itimeout`** and **`igettimeout`** functions. The use of these functions is optional, however, since the software will calculate the LAN timeout based on the SICL timeout in use and the configuration values set via the **`IO Config`** utility.

Once **`ilantimeout`** is called by the application, the automatic LAN timeout adjustment is turned off. See *Chapter 11 - SICL Language Reference* for details on the **`ilantimeout`** and **`ilangettimeout`** functions.

A timeout value of **`1`** used with the **`ilantimeout`** function has special significance, causing the LAN client to not wait for a response from the LAN server. However, the timeout value of **`1`** should be used only in special circumstances and should be used with extreme caution. For more information about this timeout value, see the **`ilantimeout`** function in *Chapter 11 - SICL Language Reference*.

Default LAN Timeout Values

The **`IO Config`** utility specifies two timeout-related configuration values for the LAN software. These values are used by the software to calculate timeout values if the application has not previously called **`ilantimeout`**.

Server Timeout	Timeout value passed to the server when an application either uses the SICL default timeout value of infinity or sets the SICL timeout to infinity (0). Value specifies the number of seconds the server will wait for the operation to complete before returning **I_ERR_TIMEOUT**. A value of **0** in this field will cause the server to be sent a value of infinity if the client application also uses the SICL default timeout value of infinity or sets the SICL timeout to infinity (**0**).
Client Timeout Delta	Value added to the SICL timeout value (server's timeout value) to determine the LAN timeout value (client's timeout value). Value specifies the number of seconds.

Once **`ilantimeout`** is called, the software no longer sends the Server Timeout value to the server and no longer attempts to determine a reasonable client-side timeout. It is assumed that the application itself wants *full* control of timeouts, both client and server. Also, **`ilantimeout`** is *per process*. That is, all sessions going out over the network are affected when **`ilantimeout`** is called. If the application has *not* called the **`ilantimeout`** function, timeouts are adjusted via the following algorithm:

- The SICL timeout, which is sent to the server, for the current call is adjusted if it is currently infinity (**0**). In that case it will be set to the Server Timeout value.
- The LAN timeout is adjusted if the SICL timeout plus the Client Timeout Delta is greater than the current LAN timeout. In that case the LAN timeout will be set to the SICL timeout plus the Client Timeout Delta.
- The calculated LAN timeout only increases as necessary to meet the needs of the application, but never decreases. This avoids the overhead of readjusting the LAN timeout every time the application changes the SICL timeout.

- The first **iopen** call used to set up the server connection uses the Client Timeout Delta specified via the **IO Config** utility for portions of the **iopen** operation. The timeout value for TCP connection establishment is not affected by the Client Timeout Delta.

To change the defaults:

1. Exit any LAN applications for SICL to be reconfigured.
2. Run the **IO Config** utility. (Click the **Agilent IO Libraries Control** and then click **Run IO Config**.)
3. Change the Server Timeout and/or Client Timeout Delta value(s).
4. Restart the LAN applications for SICL.

Timeouts in Multi-threaded Applications

To manually set the client side timeout in an application using multiple threads, be aware that **ilantimeout** may itself timeout due to contention for the LAN subsystem where multiple threads in an application are simultaneously using SICL over LAN.

Thus, if multiple threads are using SICL over LAN at the same time and LAN timeouts are expected by the application, it is recommended that **ilantimeout** be called when no other LAN I/O is occurring, such as immediately after session creation (**iopen**).

The use of the **ilantimeout** No-Wait Value for certain special cases is described under the **ilantimeout** function in *Chapter 11 - SICL Language Reference*. If the no-wait value is used and multiple threads are attempting I/O over the LAN, 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.

Timeout Configurations to Be Avoided

The LAN timeout used by the client should always be set greater than the SICL timeout used by the server. This avoids the situation where the client times out while the server continues to attempt the request, potentially holding off subsequent operations from the same client. This also avoids having the server send unwanted replies to the client.

The SICL timeout used by the server should generally be less than infinity. Having the LAN server wait less than forever allows the LAN server to detect clients that have ceased operation abruptly or network problems and subsequently release resources associated with those clients, such as locks.

Using the smallest possible value for your application will maximize the server's responsiveness to dropped connections, including the client application being terminated abnormally. Setting a value less than infinity is done by setting the Server Timeout configuration value via the **`IO Config`** utility.

Even if your application uses the SICL default of infinity or if **`itimeout`** is used to set the timeout to infinity, by setting the Server Timeout value to some reasonable number of seconds, the server will be allowed to timeout and detect network trouble and release resources.

Application Terminations and Timeouts

If an application is stopped in the middle of a SICL operation performed at the LAN server, the server continues to try the operation until the server's timeout is reached. By default, the LAN server associated with an application using a timeout of infinity that is stopped may not discover that the client is no longer running for 2 minutes. For a server other than the LAN server on HP-UX, Windows 95, Windows 98, Windows 2000, Windows NT, or the E2050, check that server's documentation for its default behavior.

If **`itimeout`** is used by the application to set a long timeout value, or if both the LAN client and LAN server are configured to use infinity or a long timeout value, the server may appear "hung". If this situation occurs, the LAN client (via the Client Timeout Delta value set with the **`IO Config`** utility) or the LAN server (via its Server Timeout value) may be configured to use a shorter timeout value.

If long timeouts must be used, the server may be reset. A LAN server may be reset by logging into the server system and stopping the LAN server that is running. The latter procedure will affect all clients connected to the server. See *Chapter 9 - Troubleshooting SICL Programs* for more details. Also, see the documentation of the server you are using for methods to reset the server.

9

Troubleshooting SICL Programs

Troubleshooting SICL Programs

This chapter contains a description of SICL error codes and provides guidelines to troubleshoot common problems with SICL. The chapter contents are:

- SICL Error Codes
- Common Windows Problems
- Common RS-232 Problems
- Common GPIO Problems
- Common LAN Problems

SICL Error Codes

When you install a default SICL error handler such as **I_ERROR_EXIT** or **I_ERROR_NOEXIT** with an **ionerror** call, a SICL internal error message will be logged. To view these messages:

- **On Windows 95 or Windows 98**, start the **Message Viewer** utility by clicking the **Agilent IO Libraries Control** (on the taskbar) and then clicking **Run Message Viewer**. You *must* start the **Message Viewer** utility before you execute a program for error messages to be logged.
- **On Windows NT or Windows 2000**, SICL logs internal messages as Windows NT events that you can view by clicking the **Agilent IO Libraries Control** (on the taskbar) and then clicking **Run Event Viewer**. Both system and application messages can be logged to the **Event Viewer** from SICL. SICL messages are identified by **SICL LOG** or by the driver name (such as **hp341i32** for the GPIB driver).

For C programs, you can use **ionerror** to install a custom error handler. The error handler can call **igeterrstr** with the given error code and the corresponding error message string will be returned. See *Chapter 3 - Programming with SICL* for more information on error handlers. This table summarizes 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 does not exist. Verify that the interface name is the one assigned with **IO Config**.
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**.

Error Code	Error String	Description
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**).
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.

Error Code	Error String	Description
I_ERR_NOLOCK	**Interface not locked**	An **iunlock** was specified when device was not 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 you have formatted the string properly. White space is not allowed.
I_ERR_TIMEOUT	**Timeout occurred**	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.

Common Windows Problems

Windows 95 and Windows 98	
Subsequent Execution of SICL Application Fails	If you terminate a program using the Task Manager, or if a program has an abnormal termination, some drivers may not unload from memory. This could cause subsequent attempts to execute the I/O program to fail. To recover from this situation, you must restart (reboot) Windows 95/Windows 98.
Windows NT and Windows 2000	
Program Appears to Hang and Cannot Be Stopped	Check that an **`itimeout`** value has been set for all SICL sessions in your program. Otherwise, when an instrument does not respond to a SICL read or write, SICL will wait indefinitely in the SICL kernel access routine, preventing the application from being stopped. To stop the application, click the “toaster” button in the upper-left corner of the window and then close the window. After a few seconds, an **`End Task`** dialog box appears. Press the **`End Task`** button to stop the application.
Formatted I/O Using **`%F`** Causes Application Error	Verify **`$(cvarsdll)`** is used when compiling the application, and either **`$(guilibsdll)`** for Windows applications or **`$(conlibsdll)`** for console applications when linking your application. Also, the **`%F`** format character for **`iprintf`** only works with languages that use **`MSVCRT.DLL`**, **`MSVCRT20.DLL`**, or **`MSVCRT40.DLL`** for their run-time library. Some versions of Visual C/C++ and Borland C/C++ use their own versions of the run-time library. They cannot share global data with SICL’s version of the run-time library and, therefore, cannot use **`%F`**.

Common RS-232 Problems

Unlike GPIB, special care must be taken to ensure that RS-232 devices are correctly connected to the computer. Verifying the configuration first may save many hours of debugging time.

No Response from Instrument	Be sure the RS-232 interface is configured to match the instrument. Check the Baud Rate, Parity, Data Bits, and Stop Bits. Also, be sure you are using the correct cabling. See *Appendix F - RS-232 Cables* for more information on correct cabling. If you are sending several commands at once, try sending commands one at a time either by inserting delays or by single-stepping the program.
Data Received from Instrument is Garbled	Check the interface configuration. Install an interrupt handler in your program that checks for communication errors.
Data Lost During Large Transfers	Check: ■ Flow control setting match ■ Full/half duplex for 3-wire connections ■ Cabling is correct for hardware handshaking

Common GPIO Problems

Because the GPIO interface has such flexibility, most initial problems come from cabling and configuration. There are many configuration fields in the **IO Config** utility that must be configured for GPIO. For example, no data transfers will work correctly until the handshake mode and polarity have been correctly set. A GPIO cable can have up to 50 wires and you may need to solder your own plug to at least one end. It is important to ensure correct hardware configuration before you begin troubleshooting the software.

If you are porting an existing 98622 application, the hardware task is simplified. The cable connections are the same and many **IO Config** fields closely approximate 98622 DIP switches. For a new application, an individual with good hardware skills should become familiar with the E2075 cabling and handshake behavior. In either case, you may want to read the *Agilent E2075 GPIO Interface Card Installation Guide*.

Some GPIO-specific reasons for certain SICL errors follow. Many of these errors can also be caused by non-GPIO problems. For example, “Operation not supported” will happen on any interface if you execute **igetintfsess** with an interface ID.

Bad Address (for iopen)

This indicates **iopen** did not succeed because the specified address (symbolic name) did not correspond to a correctly configured entry in **IO Config**. If **iopen** fails, be sure the interface is properly configured. **IO Config** establishes an entry for the GPIO card in the Windows 95, Windows 98, Windows 2000, or Windows NT registry.

We strongly encourage you to let **IO Config** handle all registry maintenance for SICL. However, you can edit registry entries manually. If you manually change the registry and enter an improper configuration value, the failed **iopen** may send a diagnostic message to the **Message Viewer** (Windows 95/Windows 98) or **Event Viewer** (Windows NT/ Windows 2000).

For example:

```
HPe2074: GPIO config, bad read_clk entry
ISA card in slot #0 NOT INITIALIZED (Invalid parameter)
```

In this case, you must correct the configuration data in the registry. The recommended procedure is to use **`IO Config`**, remove the incorrect interface name, and create a **`Configured Interface`** with legal values selected from the **`IO Config`** utility's dialog boxes.

Operation Not Supported

The E2075 has several modes. Certain operations are valid in one mode, and not supported in another. Two examples are:

```
igpioctrl(id, I_GPIO_AUX, value);
```

This operation applies only to the Enhanced mode of the data port. Auxiliary control lines do not exist when the interface is in 98622 Compatibility mode.

```
igpioctrl(id, I_GPIO_SET_PCTL, 1);
```

This operation is allowed only in Standard-Handshake mode. When the interface is in Auto-Handshake mode (the default), explicit control of the PCTL line is not possible.

No Device

This error indicates PSTS checks were set for read/write operations and a false state of the PSTS line was detected. Enabling and disabling PSTS checks is done with:

```
igpioctrl(id, I_GPIO_CHK_PSTS, value);
```

If the check seems to be reporting the wrong state of the PSTS line, correct the PSTS polarity bit via the **`IO Config`** utility. If the PSTS check is functioning properly and this error is generated, some problem with the cable or the peripheral device is indicated.

Bad Parameter

If the interface is in 16-bit mode, the number of bytes requested in an **`iread`** or **`iwrite`** function must be an even number. Although you probably view 16-bit data as words, the syntax of **`iread`** and **`iwrite`** requires a length specified as bytes.

Common LAN Problems

NOTE

Both the LAN client and LAN server may log messages to the **Message Viewer** (Windows 95/Windows 98) or **Event Viewer** (Windows NT/ Windows 2000) under certain conditions, whether or not an error handler has been registered.

General Troubleshooting Techniques

Before SICL over LAN can function, the client must be able to talk to the server over the LAN. You can use the following techniques to determine if the problem is a general network problem or is specific to the LAN software provided with SICL.

Using the `ping` Utility

If the application cannot open a session to the LAN server for SICL, the first diagnostic to try is the **ping** utility. This utility allows you to test general network connectivity between client and server machines.

Using **ping** looks something like the following, where each line after the **Pinging** line is an example of a packet successfully reaching the server.

```
>ping instserv.hp.com
Pinging instserv.hp.com[128.10.0.3] with 32 bytes of data:
Reply from 128.10.0.3:bytes=32 time=10ms TTL=255
Reply from 128.10.0.3:bytes=32 time=10ms TTL=255
Reply from 128.10.0.3:bytes=32 time=10ms TTL=255
Reply from 128.10.0.3:bytes=32 time=10ms TTL=225
```

However, if **ping** cannot reach the host, a message similar to the following is displayed that indicates the client was unable to contact the server. In this case, you should contact your network administrator to determine if there is a LAN problem. When the LAN problem has been corrected, you can retry your SICL application over LAN.

```
Pinging instserv.hp.com[128.10.0.3] with 32 bytes of data:
Request timed out.
Request timed out.
Request timed out.
Request timed out.
```

Using the `rpcinfo` Utility

Another tool you can use to determine where a problem might reside is the **`rpcinfo`** utility on an HP-UX workstation or other UNIX workstation. This tool tests whether a client can make an RPC call to a server.

rpcinfo -p. The first **`rpcinfo`** option to try is **`-p`**, which will print a list of registered programs on the server:

```
> rpcinfo -p instserv
program verses proto   port
100001    1   udp   1788  rstatd
100001    2   udp   1788  rstatd
100001    3   udp   1788  rstatd
100002    1   udp   1789  rusersd
100002    2   udp   1789  rusersd
395180    1   tcp   1138
395183    1   tcp   1038
```

Several lines of text may be returned, but the ones of interest for this example are the lines for programs **`395180`** for the SICL LAN Protocol and **`395183`** for the TCP/IP Instrument Protocol (the port numbers will vary). If the line for program **`395180`** or **`395183`** is not present, your LAN server is likely misconfigured. Consult your server's documentation, correct the configuration problem, and then retry your application.

rpcinfo -t. The second **`rpcinfo`** option which can be tried is **`–t`**, which will attempt to execute procedure 0 of the specified program. Lines similar to the following should be displayed. If not, the server is likely misconfigured or not running. Consult your server's documentation, correct the problem, and then retry your application. See the **`rpcinfo(1M)`** man page for more information.

For the SICL LAN Protocol:

```
> rpcinfo -t instserv 395180
program 395180 version 1 ready and waiting
```

For the TCP/IP Instrument Protocol:

```
> rpcinfo -t instserv 395183
program 395183 version 1 ready and waiting
```

LAN Client Problems

`iopen` Fails - Syntax Error

In this case, **`iopen`** fails with the error **`I_ERR_SYNTAX`**. If using the "lan,net_address" format, ensure that the net_address is a hostname, not an IP address. If you must use an IP address, specify the address using the bracket notation, **`lan[128.10.0.3]`**, rather than the comma notation **`lan,128.10.0.3`**.

`iopen` Fails - Bad Address

An **`iopen`** fails with the error **`I_ERR_BADADDR`**, and the error text is **`core connect failed: program not registered`**. This indicates the LAN server for SICL has not registered itself on the server machine. This may also be caused by specifying an incorrect hostname. Ensure that the hostname or IP address is correct and, if so, check the LAN server's installation and configuration.

`iopen` Fails - Unrecognized Symbolic Name

The **`iopen`** fails with the error **`I_ERR_SYMNAME`**, and the error text is **`bad hostname, gethostbyname() failed`**. This indicates the hostname used in the **`iopen`** address is unknown to the networking software. Ensure that the hostname is correct and, if so, contact your network administrator to configure your machine to recognize the hostname. The **`ping`** utility can be used to determine if the hostname is known to your system. If **`ping`** returns with the error **`Bad IP address`**, the hostname is not known to the system.

`iopen` Fails - Timeout

An **`iopen`** fails with a timeout error. Increase the Client Timeout Delta configuration value via the **`IO Config`** utility. See *Chapter 8 - Using SICL with LAN* for more information.

`iopen` Fails - Other Failures

An **`iopen`** fails with some error other than those already mentioned. Try the steps at the beginning of this section to see if the client and server can talk to one another over the LAN. If the **`ping`** and **`rpcinfo`** procedures work, check any server error logs that may be available for further clues. Check for possible problems such as a lack of resources at the server (memory, number of SICL sessions, etc.).

I/O Operation Times Out

An I/O operation times out even though the timeout being used is infinity. Increase the Server Timeout configuration value via the **`IO Config`** utility. Also, ensure the LAN client timeout is large enough if **`ilantimeout`** is used. See *Chapter 8 - Using SICL with LAN* for more information.

Operation Following a Timed Out Operation Fails

An I/O operation following a previous timeout fails to return or takes longer than expected. Ensure the LAN timeout being used by the system is sufficiently greater than the SICL timeout being used for the session in question. The LAN timeout should be large enough to allow for the network overhead in addition to the time that the I/O operation may take.

If **`ilantimeout`** is used, you must determine and set the LAN timeout manually. Otherwise, ensure the Client Timeout Delta configuration value is large enough (via the **`IO Config`** utility). See *Chapter 8 - Using SICL with LAN* for more information.

`iopen` Fails or Other Operations Fail Due to Locks

An **`iopen`** fails due to insufficient resources at the server or I/O operations fail because some other session has the device or interface locked. LAN server connections for SICL from previous clients may not have terminated properly. Consult your server's troubleshooting documentation and follow the instructions for cleaning up any previous server processes.

LAN Server Problems

SICL LAN Application Fails - RPC Error

After starting the LAN server, a SICL LAN application fails and returns a message similar to the following:

```
RPC_PROG_NOT_REGISTERED
```

There is a short (approximately 5 second) delay between starting the LAN server and the LAN server being registered with the Portmapper. Try running the SICL LAN application again.

`rpcinfo` Does Not List **`395180`** or **`395183`**

A **`rpcinfo`** query fails to indicate that program **`395180`** (SICL LAN Protocol) or **`395183`** (TCP/IP Instrument Protocol) is available on the server. If you have not yet started the LAN server, do so now. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for details to start the LAN server. If you have started the LAN server, try **`rpcinfo`** again after a few seconds to ensure the LAN server had time to register itself.

iopen Fails

An **iopen** fails when you run your application, but **rpcinfo** indicates the LAN server is ready and waiting. Ensure the requested interface has been configured on the server. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on using **IO Config** to configure interfaces for SICL.

LAN Server Appears "Hung"

The LAN server appears "hung" (possibly due to a long timeout being set by a client on an operation that will never succeed). Login to the LAN server and stop the hung LAN server process. To stop the LAN server, see the *Agilent IO Libraries Installation and Configuration Guide for Windows*.

This action will affect all connected clients, even those that may still be operational. If informational logging has been enabled using the **IO Config** utility, connected clients can be determined by log entries in the **Message Viewer** (Windows 95/Windows 98) or **Event Viewer** (Windows NT/ Windows 2000) utility.

rpcinfo Fails - can't contact portmapper

An **rpcinfo** returns the message **rpcinfo: can't contact portmapper: RPC_SYSTEM_ERROR - Connection refused**.
If the LAN server is not running, start it. If the LAN server is running, stop the currently running LAN server and then restart it.

Use **Ctrl**+**Alt**+**Del** to display a task list. Ensure that both **LAN Server** and **Portmap** are not running before restarting the LAN server. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for details to start and stop the LAN server.

rpcinfo Fails - program 395180 is not available

An **rpcinfo -t** *server_hostname* **395180 1** returns the following message:

```
rpcinfo: RPC_SYSTEM_ERROR - Connection refused
program 395180 version 1 is not available
```

Ensure that the LAN server program is running on the server.

Mouse "Hung" When Stopping LAN Server

If. after attempting to stop a LAN server via either **Ctrl**+**C** or the Windows 95,Windows 98, Windows 2000, or Windows NT x-button (in the upper-right hand corner of the window), the mouse may appear to be "hung". Press any keyboard key and the LAN server will stop and the mouse will again be operational.

10

More SICL Example Programs

More SICL Example Programs

This chapter contains two example programs that give guidelines to help you develop SICL applications. The chapter contents are:

- Example: Oscilloscope Program (C)
- Example: Oscilloscope Program (Visual Basic)

Example: Oscillosope Program (C)

This C example programs an oscilloscope (such as an Agilent 54601), uploads the measurement data, and instructs the oscilloscope to print its display to a ThinkJet printer. This program uses many SICL features and illustrates some important C and Windows programming techniques for SICL.

Program Files

The oscilloscope example files are located in the **C\SAMPLES\SCOPE** subdirectory under the SICL base directory. The subdirectory contains the source program and a number of files to help you build the example with specific compilers, depending on the Windows environment used.

SCOPE.C	Example program source file.
SCOPE.H	Example program header file.
SCOPE.RC	Example program resource file.
SCOPE.DEF	Example program module definitions file.
SCOPE.ICO	Example program icon file.
VCSCP32.MAK	Windows 95, Windows 98, Windows 2000, or Windows NT project file for Microsoft Visual C++.
BCSCP32.IDE	Windows 95, Windows 98, Windows 2000, or Windows NT project file for Borland C Integrated Development Environment.

Building the Project File

This section shows how to create the project file for this example using Microsoft Visual C. You can also load the makefile directly from the **C\SAMPLES\SCOPE** subdirectory, if you desire. If you are using another language tool, choose the appropriate project file or makefile from the **C\SAMPLES\SCOPE** subdirectory. To compile and link the example program with Microsoft Visual C:

1. Select **File | New** from the menu and select **Project** from the list box that appears. Then click **OK**.

2. The **New Project** dialog box is now displayed. Type the name you want for the project in the edit box labeled **Project Name**. Then, select **Application** from the **Project Type** list box. Select the directory location for the project in the **Directory** list box and click the **Create** button.

3. The **Project Files** dialog box is now displayed. Double-click the source files **scope.c**, **scope.rc**, and **scope.def** to add them to the project. Also add **sicl32.lib** from the SICL **C** directory. Then, click the **Close** button.

4. Select **Project | Settings** from the menu and click the **C\C++** button. Select **Code Generation** from the **Category** list box. Then, select **Multithreaded Using DLL** from the **Use Run-Time Library** list box and click **OK**.

5. Select **Tools | Options** from the menu and click the **Directories** button in the **Options** dialog box. Select **Include Files** from the **Show Directories for:** list box and click the **Add** button. Then, type **\SICL\C** and click **OK**.

6. Select **Project | Build** to build the application.

If there are no errors reported, you can execute the program by selecting **Project | Execute**. An application window will open. Several commands are available from the **Actions** menu, and any results or output will be printed in the program window. To end the program, select **File | Exit** from the program menu.

Program Overview

> **NOTE**
>
> You may want to view the program with an editor as you read through this section. The entire program is not listed here because of its length. This illustrates specific SICL features and programming techniques and is not meant to be a robust Windows application. See *Chapter 11 - SICL Language Reference* or the SICL online **`Help`** for detailed information on the SICL features used in this program.

Custom Error Handler

The oscilloscope program defines a custom error handler that is called whenever an error occurs during a SICL call. The handler is installed using **`ionerror`** before any other SICL function call is made, and will be used for all SICL sessions created in the program.

```
void SICLCALLBACK my_err_handler(INST id, int error)
{
   ...
   sprintf(text_buf[num_lines++],
     “session id=%d, error = %d:%s”, id, error,
       igeterrstr(error));
   sprintf(text_buf[num_lines++], “Select ‘File | Exit’
       to exit program!”);
   ...
// If error is from scope, disable I/O actions by
       graying out menu picks.
   if (id == scope) {
     ... code to disallow further I/O requests from user
     }
   }
```

The error number is passed to the handler, and **`igeterrstr`** is used to translate the error number into a more useful description string. If desired, different actions can be taken depending on the particular error or **`id`** that caused the error.

Locks

SICL allows multiple applications to share the same interfaces and devices. Different applications may access different devices on the same interface, or may alternately access the same device (a shared resource). If your program will be executing along with other SICL applications, you may want to prevent another application from accessing a particular interface or device during critical sections of your code. SICL provides the **`ilock/iunlock`** functions for this purpose.

```
void get_data (INST id)
{
   ...  non-SICL code

/* lock device to prevent access from other applications */
   ilock(scope);

   ...
   SICL I/O code to program scope and get data

/* release the scope for use by other applications */
   iunlock(scope);

   ...  non-SICL code
}
```

Lock the interface or device with **`ilock`** before critical sections of code, and release the resource with **`iunlock`** at the end of the critical section. Using **`ilock`** on a device session prevents any other device session from accessing the particular device. Using **`ilock`** on an interface session prevents any other session from accessing the interface and any device connected to the interface.

See **`isetlockwait`** in *Chapter 11 - SICL Language Reference* to determine actions that can be taken when a SICL call in your code attempts to access a resource that is locked by another session.

Formatted I/O

SICL provides extensive formatted I/O functionality to help facilitate communication of I/O commands and data. The example program uses a few of the capabilities of the **`iprintf/iscanf/ipromptf`** functions and their derivatives.

The **iprintf** function is used to send commands. As with all of the formatted I/O functions, the data is actually buffered. In this call, the **\n** at the end of the format:

```
iprintf(id,":waveform:preamble?\n");
```

causes the buffer to be flushed and the string to be output. If desired, several commands can be formatted before being sent and then all commands outputted at once. The formatted I/O buffers are automatically flushed whenever the buffer fills (see **isetbuf**) or when an **iflush** call is made.

When reading data back from a device, the **iscanf** function is used. To read the preamble information from the oscilloscope, use the format string "**%,20f\n**":

```
iscanf(id,"%,20f\n",pre);
```

This string expects to input 20 comma-separated floating point numbers into the **pre** array.

To upload the oscilloscope waveform data, use the string "**%#wb\n**". The **wb** indicates that **iscanf** should read word-wide binary data. The # preceding the data modifer tells **iscanf** to get the maximum number of binary words to read from the next parameter (**&elements**):

```
iscanf(id,"%#wb\n",&elements,readings);
```

The read will continue until an EOI indicator is received or the maximum number of words have been read.

Interface Sessions

Sometimes it may be necessary to control the GPIB bus directly instead of using SICL commands. This is accomplished using an interface session and interface-specific commands. This example uses **igetintfsess** to get a session for the interface to which the oscilloscope is connected. (If you know which interface is being used, it is also possible to just use an **iopen** call on that interface.)

Then, **igpibsendcmd** is used to send some specific command bytes on the bus to tell the printer to listen and the oscilloscope to send its data. The **igpibatnctl** function directly controls the state of the ATN signal on the bus.

```
void print_disp (INST id)
{
   INST hpibintf ;
   ...

   hpibintf = igetintfsess(id);
   ...

   /* tell oscilloscope to talk and printer to listen
      the listen command is formed by adding 32 to the
      device address of the device to be a listener.
      The talk command is formed by adding 64 to the
      device address ofthe device to be a talker. */

   cmd[0] = (unsigned char)63 ;   /* 63 is unlisten */
   cmd[1] = (unsigned char)(32+1) ;  /* printer at addr 1,
                                        make it a listener */
   cmd[2] = (unsigned char)(64+7) ; /* scope at addr 7,
                                        make it a talker */
   cmd[3] = '\0';            /* terminate the string */

   length = strlen (cmd) ;

   igpibsendcmd(hpibintf,cmd,length);
   igpibatnctl(hpibintf,0);

   ...
}
```

SRQs and **iwaithdlr**

Many instruments are capable of using the service request (SRQ) signal on the GPIB bus to signal the controller that an event has occurred. If an application needs to respond to SRQs, an SRQ handler must be installed with the **ionsrq** call. All SRQ handlers are called whenever an SRQ occurs.

In the example handler, the oscilloscope status is read to verify that the oscilloscope asserted SRQ, and then the SRQ is cleared and a status message is displayed. If the oscilloscope did not assert SRQ, the handler prints an error message.

```
void SICLCALLBACK my_srq_handler(INST id)
{
   unsigned char status;

   /* make sure it was the scope requesting service */
   ireadstb(id,&status);

   if (status &= 64) {
      /* clear the status byte so the scope can assert
         SRQ again if needed. */
      iprintf(id,"*CLS\n");

      sprintf(text_buf[num_lines++],
         "id = %d, SRQ received!, stat=0x%x", id,status);
   } else {
      sprintf(text_buf[num_lines++],
         "SRQ received, but not from the scope");
   }
   InvalidateRect(hWnd, NULL, TRUE);
}
```

In the routine that commands the oscilloscope to print its display, the oscilloscope is set to assert SRQ when printing is finished. While the oscilloscope is printing, the example program has the application suspend execution. SICL provides the function **`iwaithndlr`** that will suspend execution and wait until either an event occurs that would call a handler, or a specified timeout value is reached.

In the example, interrupt events are turned off with **`iintroff`** so that all interrupts are disabled while interrupts are being set up. Then, the SRQ handler is installed with **`ionsrq`**. Code to program the oscilloscope to print and send an SRQ is next, then the call to **`iwaithdlr`**, with a timeout value of 30 seconds. When the oscilloscope finishes printing and sends the SRQ, the SRQ handler will be executed and then **`iwaithdlr`** will return. A call to **`iintron`** re-enables interrupt events.

```
void print_disp (INST id)
{
   ...

   iintroff();
   ionsrq(id,my_srq_handler);/* Not supported on 82335 */

   /* tell the scope to SRQ on 'operation complete' */
   iprintf(id,"*CLS\n");
```

```
   iprintf(id,"*SRE 32 ; *ESE 1\n") ;

   /* tell the scope to print */
   iprintf(id,":print ; *OPC\n") ;

   ... code to tell the scope to print

   /* wait for SRQ before continuing program */

   iwaithdlr(30000L);
   iintron();

   sprintf (text_buf[num_lines++],"Printing complete!") ;
   ...
}
```

Example: Oscillosope Program (Visual Basic)

This Visual Basic example program uses SICL to get and plot waveform data from an Agilent 54601A (or compatible) oscilloscope. This routine is called each time the **cmdGetWaveform** command button is clicked.

Program Files

The oscilloscope example files are located in the **VB\SAMPLES\SCOPE** subdirectory under the SICL base directory. The files are:

SCOPE.FRM	Visual Basic source for the SCOPE example program.
SCOPE.MAK	Visual Basic project file for the SCOPE example program.

Loading and Running the Program

Follow these steps to load and run the **SCOPE** sample program:

1. Connect an Agilent 54601A oscilloscope to your interface.
2. Run Visual Basic.
3. Open the project file **SCOPE.MAK** by selecting **File | Open Project** from the Visual Basic menu.
4. Edit the **SCOPE.FRM** file to set the **scope_address** constant to the address of your oscilloscope. To do this:
   a. Select **Window | Procedures** from the Visual Basic menu. A View Procedure dialog box will appear.
   b. Select **SCOPE.FRM** from the Modules list box and **declarations)** from the Procedures list box. Then, click **OK**.
   c. Edit the following line so the address is set to the address of the oscilloscope:

```
>> Const scope_address = “hpib7,7”
```

5. Run the program by pressing the **F5** key or the **RUN** button on the Visual Basic Toolbar.

6. Press the **Waveform** button to get and display the waveform.

7. Press the **Integral** button to calculate and display the integral.

After performing these steps, you can create a standalone executable (**.EXE**) version of this program by selecting **File | Make EXE File** from the Visual Basic menu.

Program Overview

NOTE

You may want to view the program with an editor as you read through this section. The entire program is not listed here because of its length. This illustrates specific SICL features and programming techniques and is not meant to be a robust Windows application. See *Chapter 11 - SICL Language Reference* or the SICL online **Help** for detailed information on the SICL features used in this program.

cmdGetWaveform_Click

Subroutine that is called when the **cmdGetWaveform** command button is pressed. The command button is labeled **Waveform**.

On Error

This Visual Basic statement enables an error handling routine within a procedure. In this example, an error handler is installed starting at label **ErrorHandler** within the **cmdOutputCmd_Click** subroutine. The error handling routine is called any time an error occurs during the processing of the **cmdGetWaveform_Click** procedure. SICL errors are handled in the same way that Visual Basic errors are handled with the **On Error** statement.

cmdGetWaveform.Enabled

The button that causes the **cmdGetWaveform_Click** routine to be called is disabled when code is executing inside **cmdOutputCmd_Click**. This is good programming style.

iopen

An **iopen** call is made to open a device session for the oscilloscope. The device address for the oscilloscope is in the **scope_address** string. In this example, the default address is "**hpib7,7**". The interface name "**hpib7**" is the name given to the interface with the **IO Config** utility. The bus (primary) address of the oscilloscope follows, in this case 7. You may want to change the **scope_address** string to specify the correct address for your configuration.

igetintfsess

igetintfsess is called to return an interface session *id* for the interface to which the oscilloscope instrument is connected. This interface session will be used by the following **iclear** call to send an interface clear to reset the interface.

iclear

The **iclear** function is called to reset the interface.

itimeout

itimeout is called to set the timeout value for the oscilloscope's device session to 3 seconds.

ivprintf

The **ivprintf** function is called four times to set up the oscilloscope and then request the oscilloscope's preamble information. In each case **Chr$(10)** is appended to the format string passed as the second argument to **ivprintf**. This tells **ivprintf** to flush the formatted I/O write buffer after writing the string specified in the format string. Also, **0&** is used to specify a NULL pointer for the third argument to **ivprintf**. A NULL pointer must be passed as the third argument since no argument conversion characters were specified in the format string for **ivprintf**.

ivscanf

The **ivscanf** function is called to read the oscilloscope's preamble information into the preamble array. The first element of the preamble array is passed as the third parameter to **ivscanf**. This passes the address of the first element of the preamble array to the **ivprintf** SICL function.

ivprintf

ivprintf is called to prompt the oscilloscope for its waveform data. Again, **Chr$(10)** is appended to the format string passed as the second argument to **ivprintf**. This tells **ivprintf** to flush the formatted I/O write buffer after writing the string specified in the format string. Also, **0&** is used to specify a NULL pointer for the third argument to **ivprintf**, since no additional arguments were specified in the format string.

ivscanf	**ivscanf** is called to read in the oscilloscope's waveform. The waveform is read in as an arbitrary block of data. The format string passed as the second parameter to **ivscanf** specifies a maximum of 4000 Integer values that can be read into the array. Also, the first element of the waveform array is passed as the third parameter to **ivscanf**. This passes the address of the first element of the waveform array to the SICL **ivscanf** function.
iclose	The **iclose** subroutine closes the **scope_id** device session for the oscilloscope as well as the **intf_id** interface session obtained with **igetintfsess**.
cmdGetWaveform.Enabled	The button that causes the **cmdGetWaveform_Click** routine to be called is re-enabled when execution inside **cmdGetWaveform_Click** is finished. This allows the program to get another waveform.
Exit Sub	This Visual Basic statement causes the **cmdGetWaveform_Click** subroutine to be exited after normal processing has completed.
errorhandler:	This label specifies the beginning of the error handler that was installed for this subroutine. This handler is called whenever a run-time error occurs.
Error$	This Visual Basic function is called to get the error message for the error.
iclose	The **iclose** subroutine is called inside the error handler to close the **scope_id** device session for the oscilloscope as well as the **intf_id** interface session obtained with **igetintfsess**.
cmdGetWaveform.Enabled	This re-enables the button that causes the **cmdGetWaveform_Click** routine to be called. This allows the program to get another waveform.
Exit Sub	This Visual Basic statement causes the **cmdGetWaveform_Click** subroutine to be exited after processing an error in the subroutine's error handler.

11

SICL Language Reference

SICL Language Reference

This chapter defines all supported SICL functions, listed in alphabetical order. The chapter includes an introduction that describes the format and content for each function, and an alphabetical listing of each function.

Introduction

Each SICL function description includes:

- C syntax and Visual Basic syntax (if the function is supported on Visual Basic)
- Complete description
- Return value(s)
- Related SICL functions

This edition describes syntax structure to program SICL applications in Visual Basic version 4.0 or later. For SICL applications written in Visual Basic versions less than version 4.0, you can port the applications to Visual Basic version 4.0 or greater. See *Appendix B - Porting to Visual Basic*. You may also want to see:

- *Appendix C - SICL Error Codes* which lists all SICL error codes.
- *Appendix D - SICL Function Summary* which summarizes supported features of each core and interface-specific SICL function.

Function Specifics

Category	Description
Session Identifiers	SICL uses **session identifiers** to refer to specific SICL sessions. The **`iopen`** function creates a SICL session and returns a session identifier. A session identifier is needed for most SICL functions. 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 indicates which sessions support that function.

Category	Description
Functions Affected by Locks	Some functions are affected by locks (see 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	Some functions are affected by timeouts (see 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)
```

Description

This function is not supported over LAN. 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

IBLOCKMOVEX

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

C Syntax

```
#include <sicl.h>

int iblockmovex (id, src_handle, src_offset, src_width,
                 src_increment, dest_handle, dest_offset,
                 dest_width, dest_increment, cnt, swap);
   INST id;
   unsigned long src_handle;
   unsigned long src_offset;
   int src_width;
   int src_increment;
   unsigned long dest_handle;
   unsigned long dest_offset;
   int dest_width;
   int dest_increment;
   unsigned long cnt;
   int swap;
```

Visual Basic Syntax

```
Function iblockmovex
(ByVal id As Integer, ByVal src_handle As Long,
ByVal src_offset as Long, ByValue src_width as Integer,
ByVal src_increment as Integer, ByVal dest_handle As Long,
ByVal dest_offset as Long, ByVal dest_width as Integer,
ByVal dest_increment as Integer, ByVal cnt As Long,
ByVal swap As Integer)
```

NOTE

Not supported over LAN. If either the *src_handle* or the *dest_handle* is NULL, the handle is assumed to be for local memory. In this case, the corresponding offset is a valid memory address.

Description

`iblockmovex` moves data (8-bit byte, 16-bit word, and 32-bit long word). from memory on one device to memory on another device. This function allows local-to-local memory copies (both *src_handle* and *dest_handle* are zero), VXI-to-VXI memory transfers (both *src_handle* and *dest_handle* are valid handles), local-to-VXI memory transfers (*src_handle* is zero, *dest_handle* is valid handle), or VXI-to-local memory transfers (*src_handle* is valid handle, *dest_handle* is zero). If a bus error occurs, unexpected results may occur.

The *id* parameter is the value returned from **`iopen`**. If the *id* parameter is zero (**0**) then all handles must be zero and all offsets must be either local memory or directly dereferencable VXI pointers.

The *src_handle* argument is the starting memory address for the source data. The *dest_handle* argument is the starting memory address for the destination data. These handles must either be valid handles returned from the **`imapx`** function (indicating valid VXI memory), or zero (**0**) indicating local memory.

Both *src_width* and *dest_width* must be the same value. They specify the width (in number of bits) of the data. Specify them as 8, 16, or 32. Offset values (*src_offset* and *dest_offset*) are generally used in memory transfers to specify memory locations. The increment parameters specify whether or not to increment memory addresses.

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.

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

IPEEKX8, IPEEKX16, IPEEKX32, IPOKEX8, IPOKEX16, IPOKEX32, IPOPFIFO, IPUSHFIFO, IDEREFPTR

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 C - SICL Error Codes 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, the error handler is not called.

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 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, as 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

IDEREFPTR

Supported Sessions: . . . . . device, interface, commander

C Syntax

```
#include <sicl.h>

int idereptr (id, handle, *value);
   INST id;
   unsigned long handle;
   unsigned char *value;
```

Visual Basic Syntax

```
Function iderefptr
(ByVal id as Integer, ByVal handle as Long,
  ByVal value as Integer)
```

Description

This function tests the handle returned by **imapx**. The *id* is the valid SICL session id returned from the **iopen** function, *handle* is the valid SICL map handle obtained from the **imapx** function.

This function sets **value* to zero (**0**) if **imap** or **imapx** returns a map handle that cannot be used as a memory pointer; you must use **ipeekx8**, **ipeekx16**, **ipeekx32**, **ipokex8**, **ipokex16**, **ipokex32**, or **iblockmovex** functions. Alternately, the function returns a non-zero value if **imapx** returns a valid memory pointer that can be directly dereferenced.

Return Value

For C programs, this function returns zero (**0**) if successful or it returns 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

IMAPX, IUNMAPX, IPEEKX8, IPEEKX16, IPEEKX32, IPOKEX8, IPOKEX16, IPOKEX32, IBLOCKMOVEX

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 written to the device.
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 GPIB).
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, 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**), 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, **`ifread`** will read from the device, fill the buffer again, etc..

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. **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;
```

Description

The **igetaddr** function returns a pointer to the address string which was passed to the **iopen** call for the session id. This function is not supported on Visual Basic.

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;
```

Description

The *igetdata* function retrieves the pointer to the data structure stored by *isetdata* associated with a session. This function is not supported on Visual Basic

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 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 C - SICL Error Codes*). This value can be used directly. The **igeterrno** function will return the last error that occurred in one of the library functions.

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 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 C - SICL Error Codes)* 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
```

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

Chapter 8 - Using SICL with LAN

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
```

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, see the **`sicl.h`** file to get the exact *lu_info* syntax. If you are programming in Visual Basic, see the **`SICL.BAS`** or **`SICL4.BAS`** file for the exact syntax. **`igetluinfo`** returns 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 orsuccessful 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);
```

Description

This function is not supported on Visual Basic. 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);
```

Description

This function is not supported on Visual Basic. 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);
```

Description

This function is not supported on Visual Basic.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, 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, ATN is set. If *atnval* is **0**, 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. 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**, 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 indicates an affirmative response means no service request.
bit 2-0	A value from 0-7 specifying GPIB line to respond on for PPOLLs.

If *cval* is equal to -**1**, the device specified by *id* is disabled from generating parallel poll responses. For commander sessions, the **igpibppollconfig** function enables and disables parallel poll responses for this device (that is, how the devices responds when the controller PPOLLs).

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 PPOLLs).

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, REN is set. If *ren* is **0**, 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. 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**.

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 determine 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)
```

Description

GPIO is *not* supported over LAN. 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, 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**), 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 E2075 interface has 16 auxiliary control lines. In 98622 Compatibility Mode, it has none. Attempting to use **I_GPIO_AUX** in 98622 Compatibility Mode results in the error: **Operation not supported**.
I_GPIO_CHK_PSTS	If the *setting* parameter is non-zero, the PSTS line is checked before each block of data is transferred. If the *setting* parameter is zero (**0**), 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 E2075 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 E2075 interface has 8 or 16 data out lines, depending on the setting specified by **igpiosetwidth**. This function changes 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**, 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, the lower 16 bits of *setting* are used. If the current data width is 8 bits, only the lower 8 bits of *setting* are used.

I_GPIO_SET_PCTL	If the *setting* parameter is non-zero, 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 *E2075 GPIO Interface Card Installation Guide*. Changes made by this function can remain in the interface hardware after your program ends. On HP-UX and Windows NT/Windows 2000, the *setting* remains until the computer is rebooted. On Windows 95/Windows 98, 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/Windows 2000, the *setting* remains until the computer is rebooted. On Windows 95/Windows 98, 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 *E2075 GPIO Interface Card Installation Guide*. Changes made by this function can remain in the interface hardware after your program ends. On HP-UX and Windows NT/Windows 2000, the *setting* remains until the computer is rebooted. On Windows 95/Windows 98, 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)
```

Description

GPIO is *not* supported over LAN. The **`igpiogetwidth`** function returns the current data width (in bits) of a GPIO interface. For the E2075 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)
```

Description

GPIO is *not* supported over LAN. The **igpiosetwidth** function is used to set the data width (in bits) of a GPIO interface. For the E2075 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. This program segment is an **iwrite** example. An 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 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)
```

Description

GPIO is not supported over LAN. 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 E2075 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 E2075 interface has either 8 or 16 data in lines, depending on the setting specified by **igpiosetwidth**. 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 E2075 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 E2075 data ports are configured in Enhanced (bi-directional) Mode. False if the ports are configured in 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**, 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 E2075 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). However, 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.

Some driver suggestions are:

- 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 ();
```

Description

This function is not supported on Visual Basic. 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. The **`iintroff`**/ **`iintron`** functions do not affect the *isetintr* values or the handlers in any way. The 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 ();
```

Description

This function is not supported on Visual Basic. The **iintron** function enables all asynchronous handlers for all sessions in the process. The **iintroff**/**iintron** functions do not affect the **isetintr** values or the handlers in any way. The default is on.

Calls to **iintroff**/**iintron** can be nested, meaning that there must be an equal number of ons and offs. This means that 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
```

Description

The **`ilangettimeout`** function stores the current LAN timeout value in *tval*. If the LAN timeout value has not been set via **`ilantimeout`**, *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 *Chapter 8 - Using SICL with LAN*.

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
```

Description

The **ilantimeout** function sets 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 that should be used. Instead, the software will 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 “Using the No-Wait Value” for more information.

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.

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.

NOTE

The **`ilantimeout`** function is per process. When **`ilantimeout`** is called, all sessions going out over the network are affected.

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. The time value is *always* rounded up to the next unit of resolution.

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.

CAUTION

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.

If the no-wait value is used in a multi-threaded application and multiple threads are attempting I/O over the LAN, 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.

To use the no-wait value:

1. Prior to the **`iwrite`** call (or any formatted I/O call that will write data) that you do not want to block waiting for the returned status from the server, call **`ilantimeout`** with a timeout value of **`1`**.

2. 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).

3. The server will send a reply to the write, even though the client will 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.

4. 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 you use a value that provides some margin for error. The timeout specified to **`ilantimeout`** is in milliseconds (rounded up to the nearest second).

5. 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).

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 *Chapter 8 - Using SICL with LAN*.

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. Placing 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 this manual 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)
```

Description

NOTE

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

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

To lock a session, ensuring exclusive use of a resource, use the **ilock** function. The *id* parameter refers to a device, interface, or commander session. If *id* refers to an interface, the entire interface is locked and other interfaces are not affected by this session.

If the *id* refers to a device or commander, only that device or commander is locked and only that session may access that device or commander. However, other devices 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, 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. Locks can be nested. So, every **`ilock`** requires a matching **`iunlock`**.

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.

This 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 */
```

This 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
```

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
```

Description

NOTE

This function is not recommended for new program development. Use **IMAPX** instead. This function is not supported over LAN.

The **imap** function maps a memory space into the 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 contains 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. Accidentally accessing non-existent memory will cause bus errors.

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 can 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.

See the *Agilent 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 may find this information under the command **`signal`** in your operating system's manuals. Visual Basic programs can perform pointer arithmetic within a single page.

Return Value

For C programs, this function returns a zero (**`0`**) if an error occurs or a non-zero number if successful. This non-zero number is the address to begin mapping memory. For Visual Basic programs, no error number is returned. Instead, the global **`Err`** variable is set if an error occurs.

See Also

IUNMAP, IMAPINFO

IMAPX

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

C Syntax

```
#include <sicl.h>

unsigned long imapx (id, mapspace, pagestart, pagecnt);
   INST id;
   int mapspace;
   unsigned int pagestart;
   unsigned int pagecnt;
```

Visual Basic Syntax

```
Function imapx
ByVal id As Integer, ByVal mapspace As Integer,
ByVal pagestart As Integer, ByVal pagecnt As Integer)
```

Description

This function is not supported over LAN. The **imapx** function returns an unsigned long number, used in other functions, that maps a memory space into the process space. The SICL **ipeek?x** and **ipoke?x** 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 contains 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.

Depending on what *iderefptr* returns, memory 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. Accidentally accessing non-existent memory will cause bus errors.

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

See the *Agilent 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 can find this information under the command **signal** in your operating system's manuals. Visual Basic programs can perform pointer arithmetic within a single page.

Return Value

For C programs, this function returns a zero (**0**) if an error occurs or a non-zero number if successful. This non-zero number is either a handle or the address to begin mapping memory. Use the **iderefptr** function to determine whether the returned handle is a valid address or a handle.

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

See Also

IUNMAPX, IMAPINFO, IDEREFPTR

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)
```

Description

This function is not supported over LAN. To determine hardware constraints on memory mappings imposed by a particular interface, use the **imapinfo** function. The *id* argument specifies a VXI interface. 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. 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).

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;
```

Description

NOTE

For Visual Basic, error handlers are installed using the Visual Basic **`On Error`** statement. See *Chapter 3 - Programming with SICL* for more information on error handling with Visual Basic.

The **`ionerror`** function is used to install a SICL error handler. Many 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 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 that 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, where the *id* argument indicates the session that generated the error and the *error* argument indicates the error that occurred. See *Appendix C - SICL Error Codes* for a description of the error codes.

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

The **INST** *id* 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. If a zero (**0**) is passed as the value of *proc*, it will remove the error handler. The error procedure could perform a *setjmp*/*longjmp* or an escape using the *try*/*recover* clauses.

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.

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 ...;
   }
}
```

Return Value

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

See 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;
```

Description

This function is not supported on Visual Basic.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. To remove the interrupt handler, pass a zero (**`0`**) in the *proc* parameter. By default, no interrupt handler is installed.

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:

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

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 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;
```

Description

This function is not supported on Visual Basic. Use the **`ionsrq`** function to notify an application when an SRQ occurs. This function installs the procedure *proc* as an SRQ handler. To remove an SRQ handler, pass a zero (**`0`**) as the *proc* parameter.

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 GPIB), 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;
```

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 using the **iopen** function. This function creates a session and returns a session identifier. The session identifier should only be passed as a parameter to other SICL functions. It is not designed to be updated manually.

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. 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 *Chapter 3 - Programming with SICL*.

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 *Chapter 3 - Programming with SICL.*

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 *Chapter 3 - Programming with SICL.*

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
```

Description

This function is not recommended for new program development. Use **IPEEKX8, IPEEKX16**, or **IPEEKX32** instead. This function is not supported over LAN.

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.

The **iwpeek** and **ilpeek** functions perform byte swapping (if necessary) so that VXI memory accesses follow correct VXI byte ordering. If a bus error occurs, unexpected results may occur.

See Also

IPOKE, IMAP

IPEEKX8, IPEEKX16, IPEEKX32

C Syntax

```
#include <sicl.h>

int ipeekx8 (id, handle, offset, *value);
   INST id;
   unsigned long handle;
   unsigned long offset;
   unsigned char *value;

int ipeekx16 (id, handle, offset, *value);
   INST id;
   unsigned long handle;
   unsigned long offset;
   unsigned short *value

int ipeekx32 (id, handle, offset, *value);
   INST id;
   unsigned long handle;
   unsigned long offset;
   unsigned long *value))
```

Visual Basic Syntax

```
Function ipeekx8
(ByVal id As Integer, ByVal handle As Long,
 ByVal offset as Long, ByVal value As Integer)
```

(syntax is the same for *ipeekx16* and *ipeekx32*)

Description

This function is not supported over LAN. The **ipeekx8, ipeekx16,** and **ipeekx32** functions read the values stored at *handle* and *offset* from memory and returns the value from that address. These functions are generally used in conjunction with the SICL **imapx** function to read data from VXI address space. The **ipeekx8** and **ipeekx16** functions perform byte swapping (if necessary) so that VXI memory accesses follow correct VXI byte ordering. If a bus error occurs, unexpected results may occur.

See Also

IPOKEX8, IPOKEX16, IPOKEX32, IMAPX

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)
```

Description

This function is not recommended for new program development. Use **`IPOKEX8, IPOKEX16`**, or **`IPOKEX32`** instead. This function is not supported over LAN. 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. *addr* is a valid memory address. *val* is a valid data value.

The **`iwpoke`** and **`ilpoke`** functions perform byte swapping (if necessary) so that VXI memory accesses follow correct VXI byte ordering. If a bus error occurs, unexpected results may occur.

See Also

IPEEK, IMAP

IPOKEX8, IPOKEX16, IPOKEX32

C Syntax

```
#include <sicl.h>

int ipokex8 (id, handle, offset, value);
   INST id;
   unsigned long handle;
   unsigned long offset;
   unsigned char value;

int ipokex16 (id, handle, offset, value);
   INST id;
   unsigned long handle;
   unsigned long offset;
   unsigned short value;

int ipokex32 (id, handle, offset, value);
   INST id;
   unsigned long handle;
   unsigned long offset;
   unsigned long value;
```

Visual Basic Syntax

```
Sub ipokex8
(ByVal id As Integer, ByVal handle As Long,
 ByVal offset as Long, ByVal value As Integer)
(syntax is the same for ipokex16 and ipokex32.)
```

Description

This function is not supported over LAN. The **ipokex8, ipokex16**, and **ipokex32** functions write to memory. The functions are generally used in conjunction with the SICL **imapx** function to write to VXI address space. *handle* is a valid memory address, *offset* is a valid memory offset. *val* is a valid data value. The **ipokex16** and **ipokex32** functions perform byte swapping (if necessary) so that VXI memory accesses follow correct VXI byte ordering. If a bus error occurs, unexpected results may occur.

See Also

IPEEKX8, IPEEKX16, IPEEKX32, IMAPX

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)
```

Description

This function is not supported over LAN. The **`i?popfifo`** functions read data from a FIFO and puts data 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. 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;
```

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**, data are sent to the device to which the **INST** refers. If the first argument is a character buffer, data are 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 that do not comply with IEEE 488.2.

Re-addressing occurs under the following circumstances. 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.:

- 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.

Restrictions Using `ivprintf` in 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&)
```

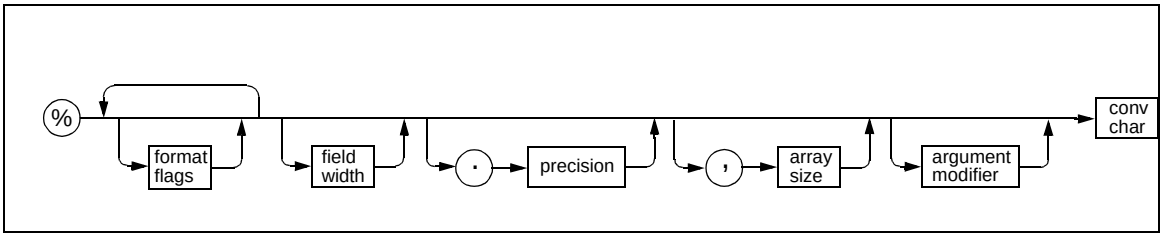
The special characters are:

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.



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 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

Some examples follow 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)
`%@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*

f*ormat 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`. `%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`. `%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`. `%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**. **%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*

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, 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/Windows 2000, corresponding *arg* is a pointer to a FILE descriptor. (Not supported on Windows 95/ Windows 98.) 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.

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.

`iprintf` can be called many times without any data being flushed to the session. There are only three 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, data will be lost). If an error handler is installed, it will be called or the error number will be set to the appropriate value.

Return Value

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

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, va_list ap);
INST id;
const char *writefmt;
const char *readfmt;
param arg1,arg2,...;
va_list ap;
```

Description

This function is not supported on Visual Basic. 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. Next, it 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. Then, 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)
```

Description

This function is not supported over LAN. 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. 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**), 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, the number of bytes read will not be returned.

To pass a NULL *reason* or *actualcnt* parameter to **`iread`** in Visual Basic, use 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 *termchr*).
- 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. Placing 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 this manual 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;
```

Visual Basic Syntax

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

Description

These functions read formatted data, convert the data, and store the results into *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 *arg*s converted is returned. The *format* string contains:

- White-space characters, which are spaces, tabs, or special characters. Use the white-space characters and conversion commands to create the *format* string's contents.
- An ordinary character (not %), which must match the next non-white-space character read from the device.
- Format conversion commands.

Notes on Using `iscanf`

Using `itermchr` with `iscanf`. The **`iscanf`** function only terminates reading on an END indicator or the termination character specified by **`itermchar`**.

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:

1. 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);
   ```

2. 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);
   ```

3. 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);
```

4. 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

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 valid 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

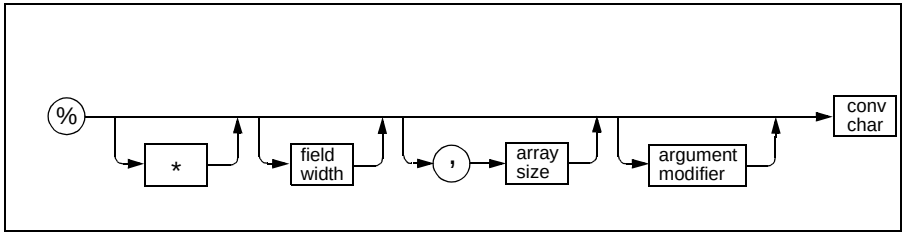
White-Space Characters for Visual Basic

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.



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, 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.

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. (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

Examples of conversion commands used in the format string and typical input data that would satisfy the conversion commands follow

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 hexadecimal 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, 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. 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. 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. 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. 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, 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, 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, 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, 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, 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, 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, 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, Windows 98, Windows 2000, 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

This table lists types of data that each numeric format accepts. Conversion types **i** and **d** and types **f** and **e**,**g** are not the same.

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

Return Value

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 sets 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. Supported baud rates are: 50, 110, 300, 600, 1200, 2400, 4800, 7200, 9600, 19200, 38400, and 57600.
I_SERIAL_PARITY	These 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	Acceptable values for *setting* are: **I_SERIAL_STOP_1** 1 stop bit **I_SERIAL_STOP_2** 2 stop bits
I_SERIAL_WIDTH	Acceptable values for *setting* are: **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	Sets 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	Acceptable values for *setting* are: **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**). Supported types of flow control are: **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	Sets the type of END Indicator to use for reads. 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 provides two different methods of indicating EOI. The first method is to use a character with 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**).

I_SERIAL_READ_EOI (cont)	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, means EOI. Valid values for the *setting* are: ■ **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 writes. The following are valid values: ■ **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, 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 finds 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** 1stop 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 Modem Status Lines that are currently being asserted. The value of the *result* parameter will be the logical OR of all of serial lines currently being asserted. The serial lines are the Modem Control Lines and the Modem Status Lines. Supported serial lines are: ■ **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, since 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_PARERR** - 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**). Supported types of flow control are: ■ **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. These 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 **0**xff. 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. These 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;
```

Description

This function is not supported on Visual Basic. **isetbuf** sets 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. 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, the buffer is *not* flushed by newline characters in the argument list.) For read buffers, the buffer is never flushed and 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. 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;
```

Description

This function is not supported on Visual Basic.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. The programmer is responsible for buffer management (buffer allocation/deallocation).

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;
```

Description

This function is not supported on Visual Basic. The **isetintr** function enables interrupt handling for a specified 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 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. .

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** and disable when *secval*=**0**. This interrupt will be generated whether or not 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.

Generic interrupts for the GPIB interface are:

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. These 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. This interrupt is enabled using *isetintr* by specifying a *secval* greater than **0**. If *secval*=**0**, this interrupt is disabled. 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, this device is addressed to listen. ■ Bit 0 = 0, this device is not addressed to listen. ■ Bit 1 = 1, this device is addressed to talk. ■ Bit 1 = 0, this device is not addressed to talk.

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 GPIO interface is always active. Therefore, the interrupts for **`I_INTR_INTFACT`** and **`I_INTR_INTFDEACT`** will never occur. 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**.

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`	The status lines that can cause this interrupt are DCD, CTS, DSR, and RI. 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.

I_INTR_SERIAL_MSL (cont)	Most implementations of the ring indicator interrupt only deliver the interrupt when the state goes from high to low (a trailing edge). This differs from other MSLs in that it is not just a state change that causes the interrupts.
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.

These are the 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**, this is disabled. The value written into the signal register is returned in the *secval* parameter of the interrupt handler.

VXI Interface Session Interrupts. These 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**, 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**, 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**, 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.

Generic interrupts for the VXI interface are:

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**, 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 "Asynchronous Events and HP-UX Signals" in the *Agilent 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 returned is **I_ERR_LOCKED**.

If *flag* is non-zero, 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**), 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.

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, an SRQ notification is given to the remote controller, if its identity is known. If bit 6 is not set, 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;
```

Description

This function is not supported on Visual Basic. The **isetubuf** function supplies 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 sets 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. Calling **isetubuf** flushes the buffer specified in the mask parameter.

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).

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. 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 **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. (These two conversions are identical.)

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. *datasize* 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, or 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. 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	Defined if the native controller is little-endian.
I_ORDER_BE	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 defines a termination character condition.

The *tchr* argument is the character specifying the termination character. If *tchr* is between 0 and 255, **iread** terminates when it reads the specified character. If *tchr* is -1, 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. The **iscanf** function terminates reading on an END indicator or the termination chqaracter specified by **itermchr**.

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. 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 sends 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. **itrigger** 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. **itrigger** 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. 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. **`itrigger`** causes one or more VXI trigger lines to fire. Trigger lines fired are 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 this manual 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. Calling the **iunlock** function may not actually unlock a device or interface again. For example, see 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)
```

Description

This function is not recommended for new program development. Use IUNMAPX instead. The function is not supported over LAN. 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

IUNMAPX

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

C Syntax

```
#include <sicl.h>

int iunmapx (id, handle, mapspace, pagestart, pagecnt);
   INST id;
   unsigned long handle;
   int mapspace;
   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)
```

Description

This function is not supported over LAN. The **iunmapx** 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

IMAPX

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 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 device is a VXI device. Returns 1 if device is a MXI device.
I_VXI_BUS_TRIGSUPP	Returns a numeric value indicating which triggers are supported. The numeric value is the sum of the following values: I_TRIG_STD 0x0000001L I_TRIG_ALL 0xffffffffL I_TRIG_TTL0 0x00001000L I_TRIG_TTL1 0x00002000L I_TRIG_TTL2 0x00004000L I_TRIG_TTL3 0x00008000L I_TRIG_TTL4 0x00010000L I_TRIG_TTL5 0x00020000L I_TRIG_TTL6 0x00040000L I_TRIG_TTL7 0x00080000L I_TRIG_ECL0 0x00100000L I_TRIG_ECL1 0x00200000L I_TRIG_ECL2 0x00400000L I_TRIG_ECL3 0x00800000L I_TRIG_EXT0 0x01000000L I_TRIG_EXT1 0x00200000L I_TRIG_EXT2 0x00400000L I_TRIG_EXT3 0x00800000L I_TRIG_CLK0 0x10000000L I_TRIG_CLK1 0x20000000L I_TRIG_CLK2 0x40000000L I_TRIG_CLK10 0x80000000L I_TRIG_CLK100 0x00000800L I_TRIG_SERIAL_DTR 0x00000400L I_TRIG_SERIAL_RTS 0x00000200L I_TRIG_GPIO_CTL0 0x00000100L I_TRIG_GPIO_CTL1 0x00000080L

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. This static data is set up by the VXI resource manager.

vxiinfo structure (C Programs)

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 */
}
```

vxiinfo structure (Visual Basic Programs)

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 Resource Manager information.

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, as shown. Any combination of values may be used in *which* by performing a bit-wise OR of the desired values. To fire trigger lines (assert, then de-assert the lines), use **ixtrig** instead of **ivxitrigon** and **ivxitrigoff.**

I_TRIG_ALL	All standard triggers for this interface (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

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 the same values as the **ixtrig** command. Any combination of values may be used in *which* by performing a bit-wise OR of the desired values.

I_TRIG_ALL	All standard triggers for this interface (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

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 listed.

I_TRIG_ALL	All standard triggers for this interface (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, thhis 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 suspends the process until the interface or device is active (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**), this function does not read a word-serial response. If *wsresp* is non-zero, 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.

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;
```

Description

This function is not supported on Visual Basic. The **iwaithdlr** function causes the process to suspend until 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, iwaithdlr terminates and generates an error if no handler executes before the given time expires. If *timeout* is zero, **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. The time value is *always* rounded up to the next unit of resolution.

The **iwaithdlr** function will implicitly enable interrupts. If you have called **iintroff**, **iwaithdlr** will re-enable interrupts and disable them again before returning.

Interrupts should be disabled with **iintroff**if **iwaithdlr** is used. The reason for disabling interrupts is that a race condition exists between the **isetintr** and **iwaithdlr**. Thus, if you only expect one interrupt, it might come before **iwaithdlr** executes. 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. This is not a defect, since the application must handle 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 sends 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. To pass a NULL *actualcnt* parameter to **iwrite** in Visual Basic, 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 sends an extended trigger to an interface. The *which* argument can be:

I_TRIG_STD	Standard trigger operation for all interfaces. **I_TRIG_STD** operation depends on the specific interface as shown in the following subsections.
I_TRIG_ALL	All standard triggers for this interface (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. The trigger lines fired are determined by the **ivxitrigroute** function. The **I_TRIG_STD** value has no default value. If **I_TRIG_STD** 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

Visual Basic programs call this routine without the initial underscore (_). This routine is called when a program is finished with all SICL I/O resources. Calling this routine is not required on Windows 95, Windows 98, Windows 2000, Windows NT, or HP-UX.

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

SICL System Information

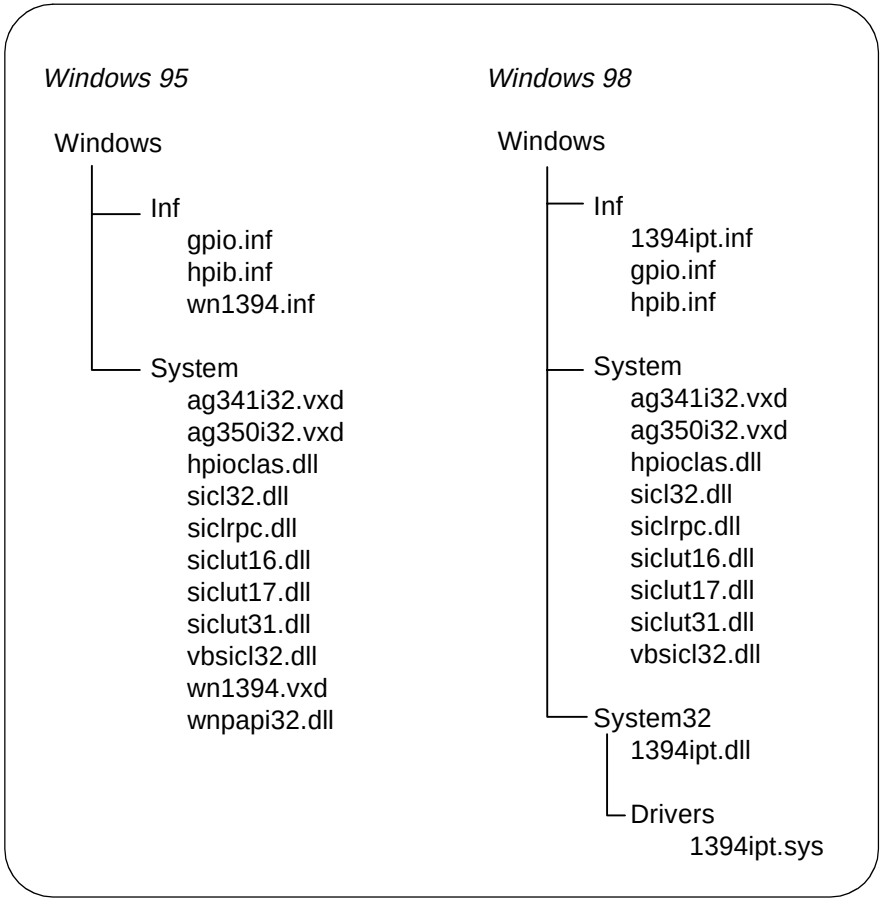
SICL System Information

This appendix provides information on SICL software files and system interaction in Windows 95, Windows 98, Windows 2000, and Windows NT. This information can be used as a reference for removing SICL from a system, if necessary.

Windows 95/Windows 98

File Location

All SICL files are installed in the base directory specified by the person who installs SICL, with the exception of several common files that Windows must be able to locate. On Windows 95 and Windows 98, the following files are copied to the Windows subdirectory.



The Registry

SICL places the following key in the Windows 95 or Windows 98 registry under HKEY_LOCAL_MACHINE:

```
Software\Agilent\IO Libraries\CurrentVersion
```

Also, if the LAN Server is configured, the following key will be created under HKEY_LOCAL_MACHINE if it did not previously exist:

```
Software\Microsoft\Windows\CurrentVersion\RunServices
```

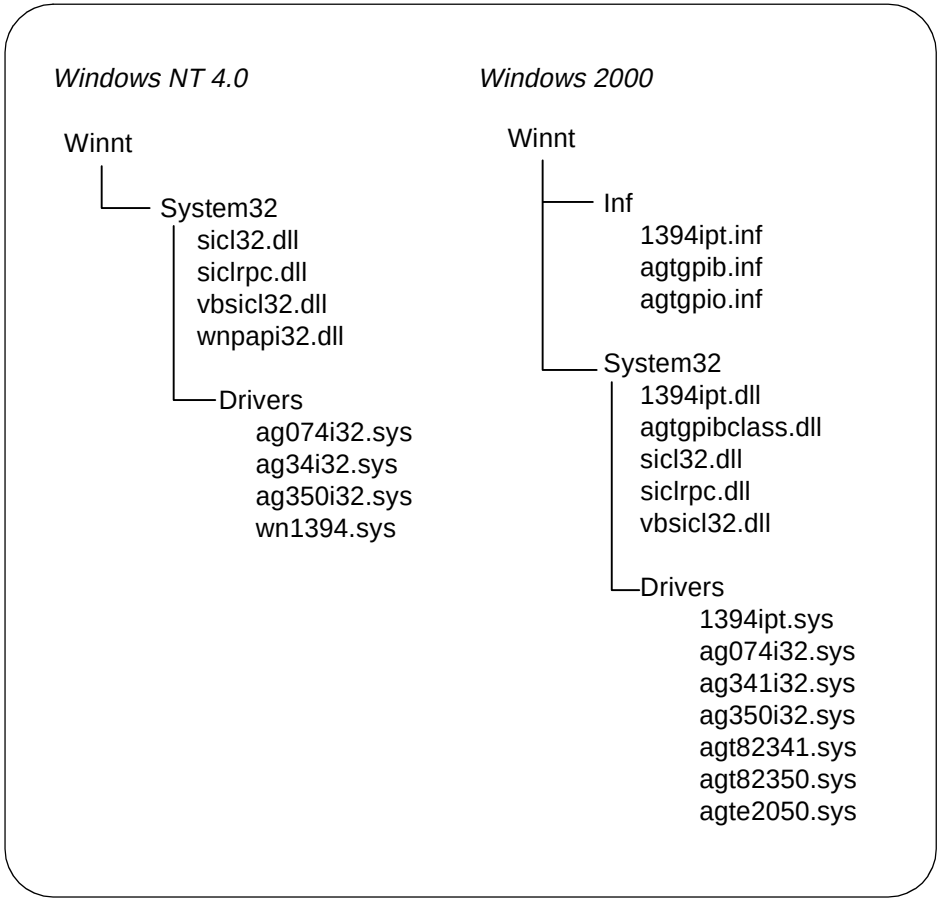
SICL Configuration Information

SICL configuration information is stored in the Windows 95 or Windows 98 registry under the `Software\Agilent\IO Libraries\CurrentVersion` branch under HKEY_LOCAL_MACHINE.

Windows NT/Windows 2000

File Location

All SICL files are installed in the base directory specified by the person who installs SICL, with the exception of several common files that Windows must be able to locate. On Windows NT and Windows 2000, the following files are copied to the Windows subdirectory.



The Registry

SICL places the following keys in the Windows NT registry under `HKEY_LOCAL_MACHINE`:

- `Software\Agilent\IO Libraries\CurrentVersion`
- `System\CurrentControlSet\Control\GroupOrderList`
- `System\CurrentControlSet\Control\ServiceOrderList`
- `System\CurrentControlSet\Services\hp341i32`
- `System\CurrentControlSet\Services\EventLog\Application\SICL Log`
- `System\CurrentControlSet\Services\EventLog\System\hp341i32`

SICL Configuration Information

SICL configuration information is stored in the Windows NT or Windows 2000 registry under the `Software\Agilent\IO Libraries\CurrentVersion` branch under `HKEY_LOCAL_MACHINE`.

B

Porting to Visual Basic

Porting to Visual Basic

This edition of this manual shows how to program SICL applications in Visual Basic version 4.0 or later. For SICL applications written in an earlier Visual Basic version than version 4.0 (for example, version 3.0), you can port your SICL applications to Visual Basic version 4.0 or later.

Porting SICL applications to Visual Basic 4.0 or later is a matter of adding the **SICL4.BAS** declaration file (rather than the **SICL.BAS** file) to each project that calls SICL for Visual Basic 4.0 or later programs. There may also be changes in functions when passing null pointers for strings to SICL functions. For example, in Visual Basic version 3.0, the preceding **ByVal** keyword was used as follows:

```
ivprintf(id, mystring, ByVal 0&)
```

In Visual Basic version 4.0 or later, you only need to pass the **0&** null pointer because version 4.0 or later knows this is by reference:

```
ivprintf(id, mystring, 0&)
```

Once you have added the **SICL4.BAS** declaration file to each project and removed `ByVal` keywords preceding null pointers for strings, your SICL applications will run correctly with Visual Basic 4.0 or later.

C

SICL Error Codes

SICL Error Codes

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** does not 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 **IO 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	CRC, Checksum, etc. 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	I/O error occurred for this communication session.
I_ERR_LOCKED	Locked by another user	Resource is locked by another session (see **isetlockwait**).
I_ERR_NESTEDIO	Nested I/O	Attempt to call another SICL function when current SICL function has not completed. 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.

Error Code	Error String	Description
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 was not 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.

Notes:

D

SICL Function Summary

SICL Function Summary

The following tables summarize supported features for each SICL function. The first table lists the core (interface-independent) SICL functions that apply to all types of devices and interfaces. The tables after that list the interface-specific SICL functions (SICL functions specific to GPIB, GPIO, LAN, RS-232/Serial, and VXI interfaces, respectively).

Each table shows if the SICL function supports device (**DEV**), interface (**INTF**), and/or commander (**CMDR**) session(s) and/or is affected by the **`ilock`** (**LOCK**) and/or the **`itimeout`** (**TIMEOUT**) function(s).

Also, the tables titled “Core SICL Functions” and “VXI SICL Functions” have the additional column, **LAN CLIENT TIMEOUT**. SICL functions with Xs in this column may timeout over LAN, even those functions that cannot timeout over local interfaces.

Core SICL Functions

Function	DEV	INTF	CMDR	LOCK	TIMEOUT	LAN CLIENT TIMEOUT
IABORT						
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

Core SICL Functions

Function	DEV	INTF	CMDR	LOCK	TIMEOUT	LAN CLIENT TIMEOUT
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
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

Core SICL Functions

Function	DEV	INTF	CMDR	LOCK	TIMEOUT	LAN CLIENT TIMEOUT
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

GPIB 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

GPIB SICL Functions

Function	DEV	INTF	CMDR	LOCK	TIMEOUT
IGPIBPPOLLCONFIG	X		X	X	X
IGPIBPPOLLRESP			X	X	X
IGPIBRENCTL		X		X	X
IGPIBSENDCMD		X		X	X
IGPIBSETT1DELAY		X		X	X

GPIO SICL Functions

Function	DEV	INTF	CMDR	LOCK	TIMEOUT
IGPIOCTRL		X		X	X
IGPIOGETWIDTH		X			
IGPIOSETWIDTH		X		X	X
IGPIOSTAT		X			

LAN SICL Functions

Function	DEV	INTF	CMDR	LOCK	TIMEOUT
IGETGATEWAYTYPE	X	X	X		
ILANGETTIMEOUT		X			
ILANTIMEOOUT		X			

RS-232/Serial 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 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

E

RS-232 Cables

RS-232 Cables

This appendix lists several general purpose RS-232 cables and adapters. Consult your instrument's operating manual for information on the status lines used for handshaking.

Cable/Adapter Part Numbers

In the following table, recommended cables and adapters are shown in **boldface** type. Other cables are listed since they may work better than the recommended cable/adapter in some applications. In the table, “a” and “b” are defined as:

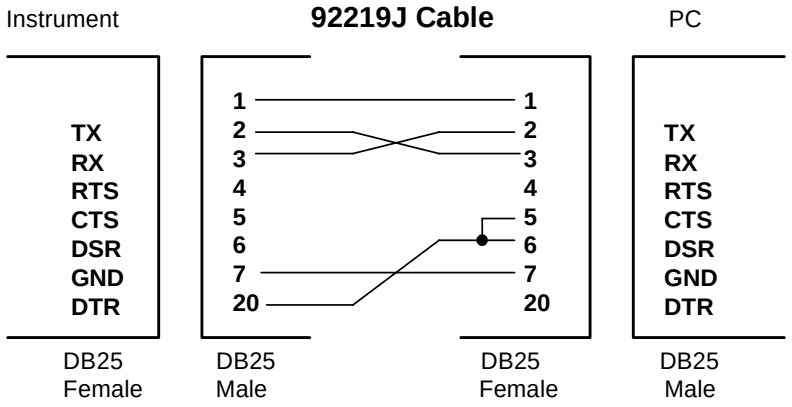
- [a] One of four adapters in the *34399A RS-232 Adapter Kit*. Kit includes four adapters to go from DB9 Female Cable (34398A) to PC/Printer DB25 Male or Female, or to modem DB9 Female or DB25 Female.
- [b] Part of *34398A RS-232 Cable Kit*. Kit comes with RS-232, 9-pin Female to 9-pin Female Null modem/printer cable and one adapter 9-pin Male to 25-pin Female (part number 5181-6641). The adapter is also located in the *34399A RS-232 Adapter Kit*.

Instrument Connector	Computer/Printer Connector	Cable Part Number	Adapter Part Number	Length
9-Pin Male	25-Pin Male	24542H 24542U **F1047-80002 [b]**	none 5181-6641 [a] **5181-6641 [a]**	3m (9ft 10in) 3m (9ft 10in) 2.5m (8ft 2.5in)
9-Pin Male	25-Pin Female	24542G 24542U **F1047-80002 [b]**	none 5181-6640 [a] **5181-6640 [a]**	3m (9ft 10in) 3m (9ft 10in) 2.5m (8ft 2.5in)
9-Pin Male	9-Pin Male	24542U 24542H & 24542M **F1047-80002 [b]**	none none none	3m (9ft 10in) 6m (19ft 10in) 2.5m (8ft 2.5in)
9-Pin Male	25-Pin Female	24542M 24542U **F1047-80002 [b]**	none 5181-6642 [a] **5181-6642 [a]**	3m (9ft 10in) 3m (9ft 10in) 2.5m (8ft 2.5in)
9-Pin Male	9-Pin Female	24542U **F1047-80002 [b]**	5181-6639 [a] **5181-6639 [a]**	3m (9ft 10in) 2.5m (8ft 2.5in)
25-Pin Female	25-Pin Female	24542G	5181-6642 [a]	3m (9ft 10in)
25-Pin Female	9-Pin Female	24542G **24542M**	5181-6639 [a] none	3m (9ft 10in) 3m (9ft 10in)
25-Pin Female	25-Pin Male	17255D C2913A **24542G**	none none **5181-6641 [a]**	1.2m (3ft 11in) 1.2m (3ft 11in) 3m (9ft 10in)

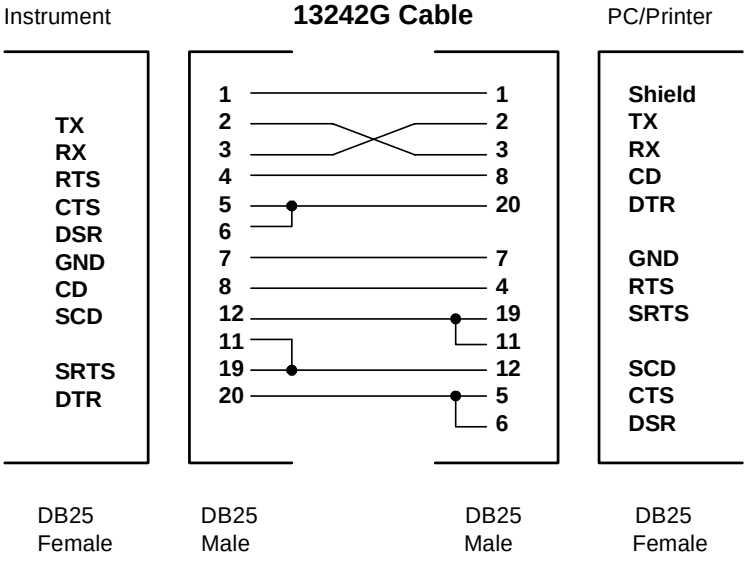
Cable/Adapter Part Numbers

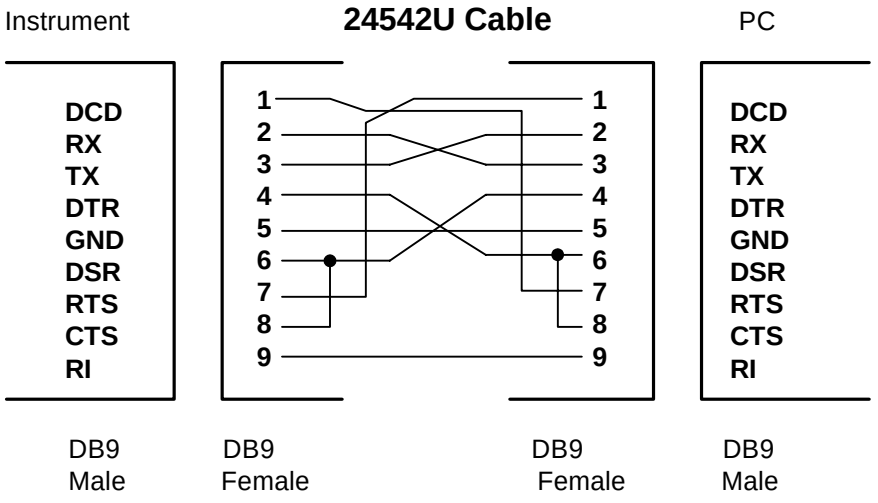
Instrument Connector	Computer/Printer Connector	Cable Part Number	Adapter Part Number	Length
25-Pin Female	25-Pin Female	13242G 17255M C2914A **24542G**	none none none **5181-6640 [a]**	5m (16ft 8in) 1.5m (4ft 11in) 1.2m (3ft 11in) 3m (9ft 10in)
25-Pin Female	9-Pin Male	**24542G** 24542U F1047-80002 [b]	none 5181-6640 [a] 5181-6640 [a]	3m (9ft 10in) 3m (9ft 10in) 2.5m (8ft 2.5in)

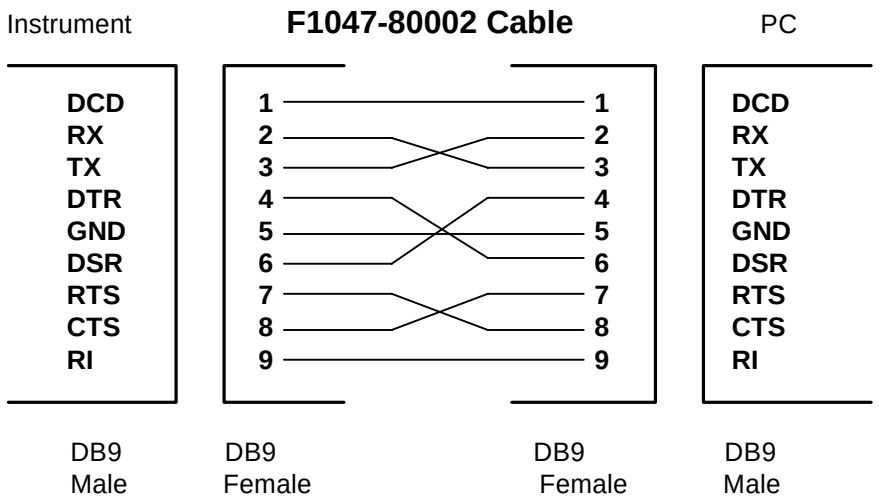
Cable/Adapter Pinouts

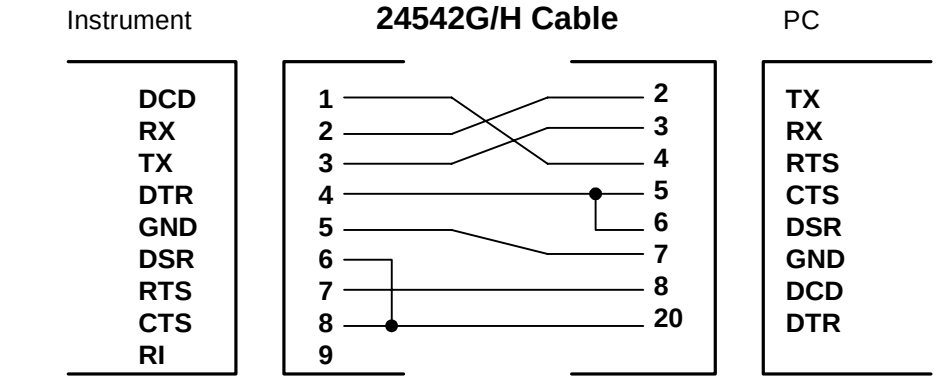


NOTE: The 92219J is directional. This cable may work differently when swapped end-to-end.

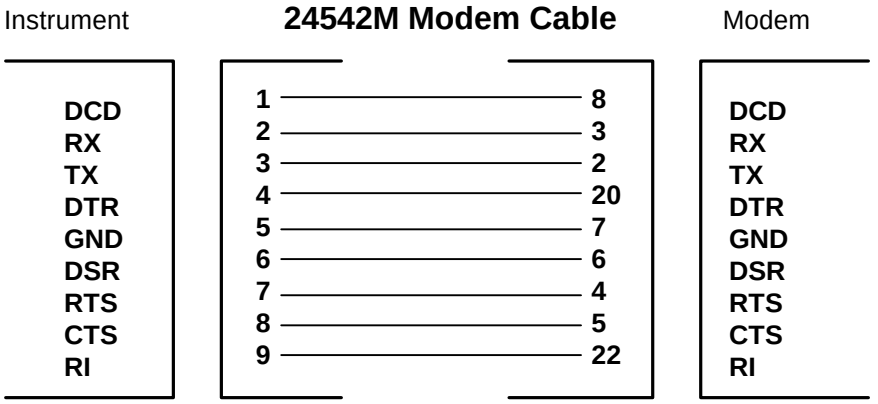



Instrument
24542U Cable
PC
DCD
RX
TX
DTR
GND
DSR
RTS
CTS
RI
1
2
3
4
5
6
7
8
9
DB9 Male
DB9 Female
DB9 Female
DB9 Male

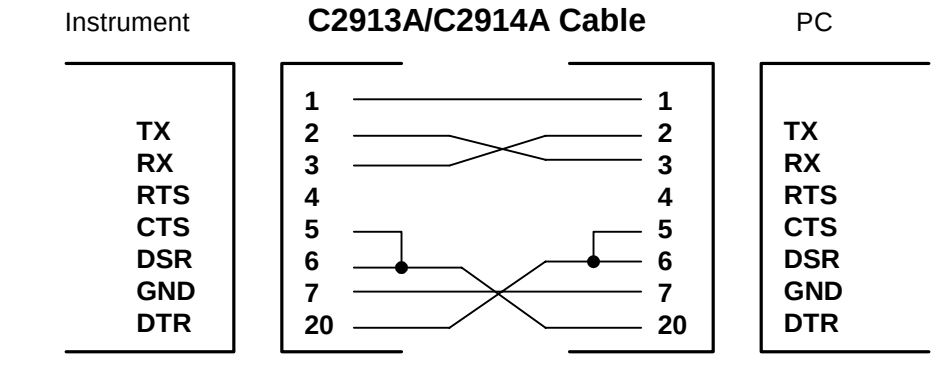

Instrument
F1047-80002 Cable
PC
DCD
RX
TX
DTR
GND
DSR
RTS
CTS
RI
1
2
3
4
5
6
7
8
9
DB9 Male
DB9 Female
DB9 Female
DB9 Male



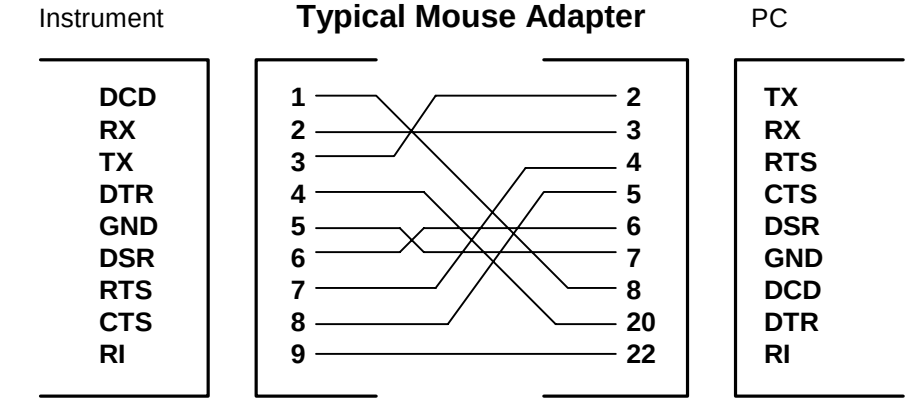
24542H	DB9 Male	DB9 Female	DB25 Female	DB25 Male
24542G	DB9 Male	DB9 Female	DB25 Male	DB25 Female



DB9 Male	DB9 Female	DB25 Male	DB25 Female

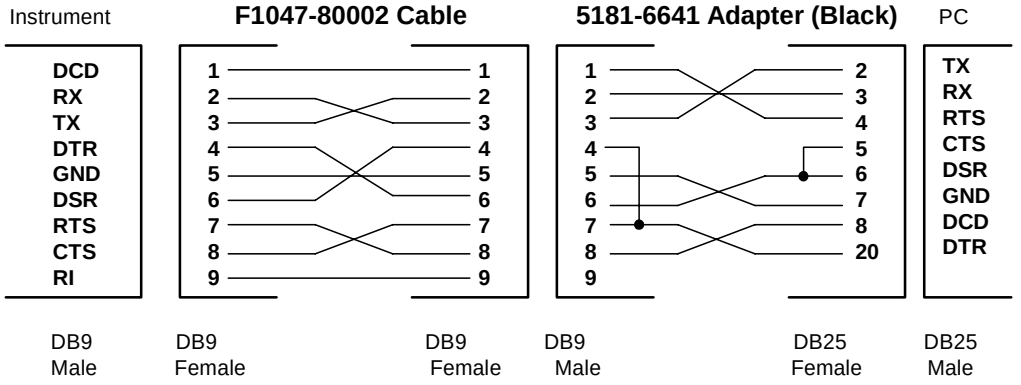


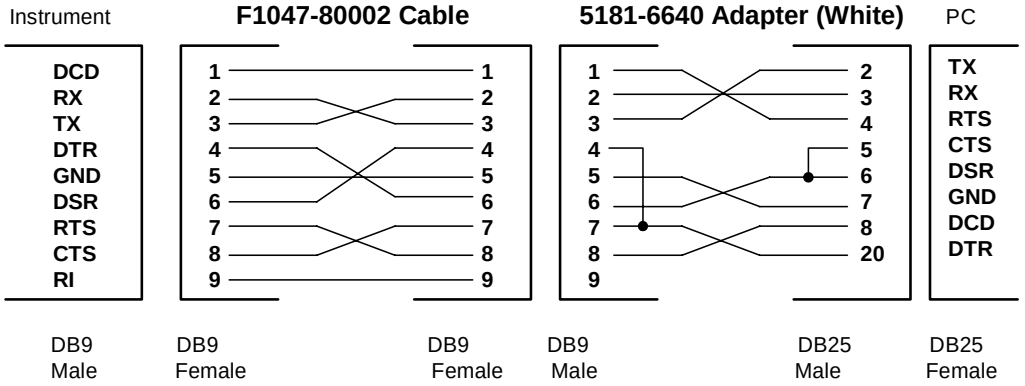
C2913A	DB25 Female	DB25 Male	DB25 Female	DB25 Male
C21914A	DB25 Female	DB25 Male	DB25 Male	DB25 Female

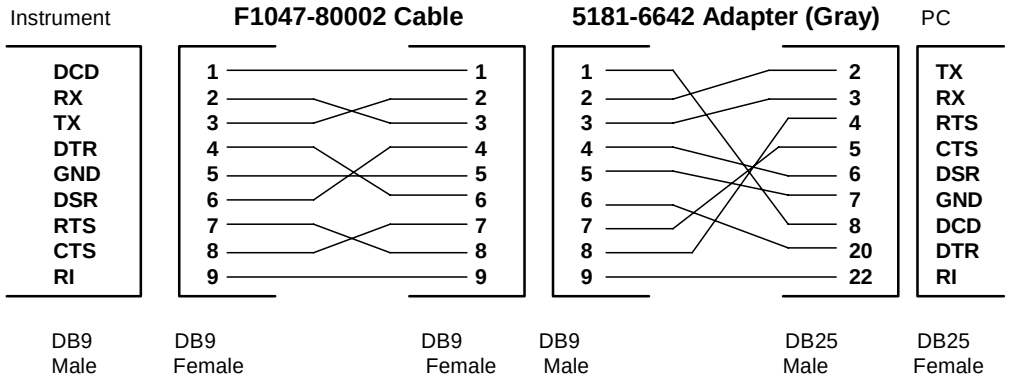


DB9 Female	DB9 Male	DB25 Female	DB25 Male

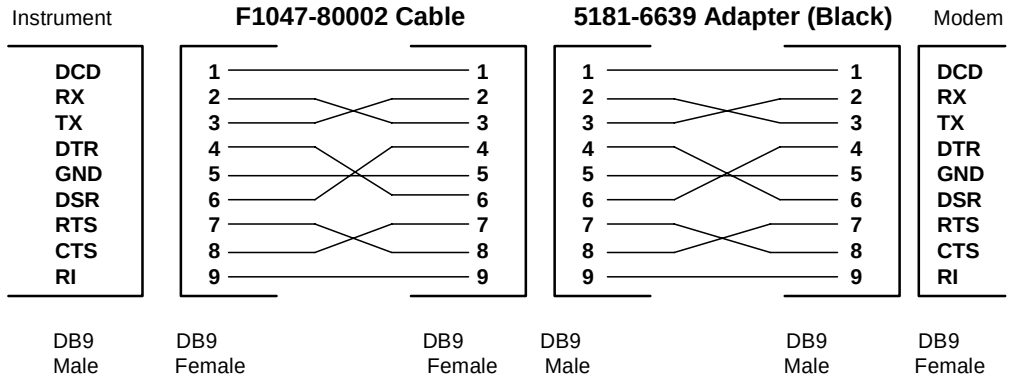
A mouse adapter works well as a 9-pin to 25-pin adapter with a PC.

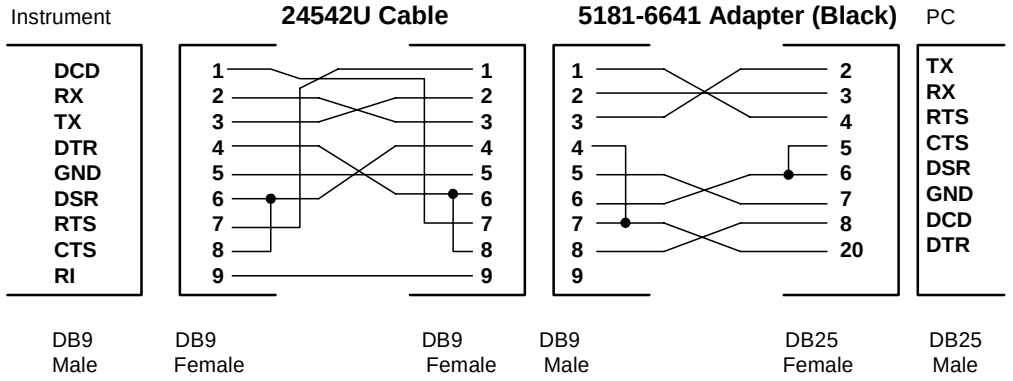

Instrument
F1047-80002 Cable
5181-6641 Adapter (Black)
PC
DCD
RX
TX
DTR
GND
DSR
RTS
CTS
RI
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 20
TX
RX
RTS
CTS
DSR
GND
DCD
DTR
DB9 Male
DB9 Female
DB9 Female
DB9 Male
DB25 Female
DB25 Male

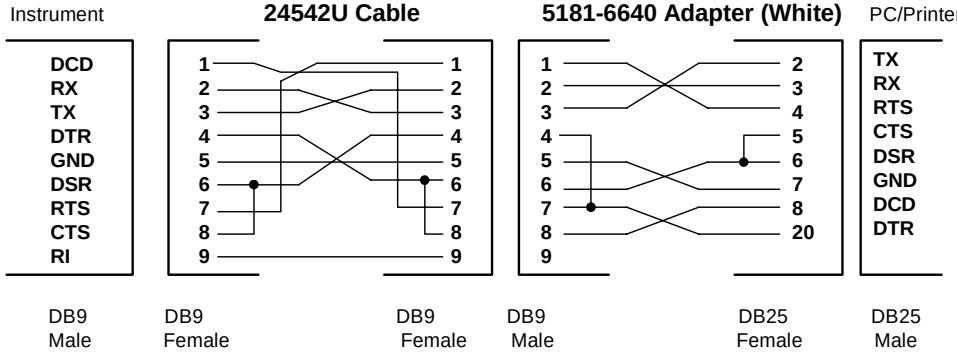

Instrument
F1047-80002 Cable
5181-6640 Adapter (White)
PC
DCD
RX
TX
DTR
GND
DSR
RTS
CTS
RI
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 20
TX
RX
RTS
CTS
DSR
GND
DCD
DTR
DB9 Male
DB9 Female
DB9 Female
DB9 Male
DB25 Male
DB25 Female

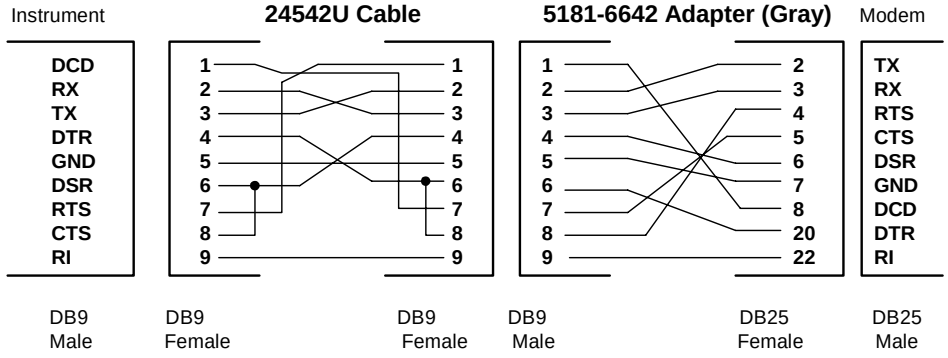

Instrument
F1047-80002 Cable
5181-6642 Adapter (Gray)
PC
DCD
RX
TX
DTR
GND
DSR
RTS
CTS
RI
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 20 22
TX
RX
RTS
CTS
DSR
GND
DCD
DTR
RI
DB9 Male
DB9 Female
DB9 Female
DB9 Male
DB25 Male
DB25 Female

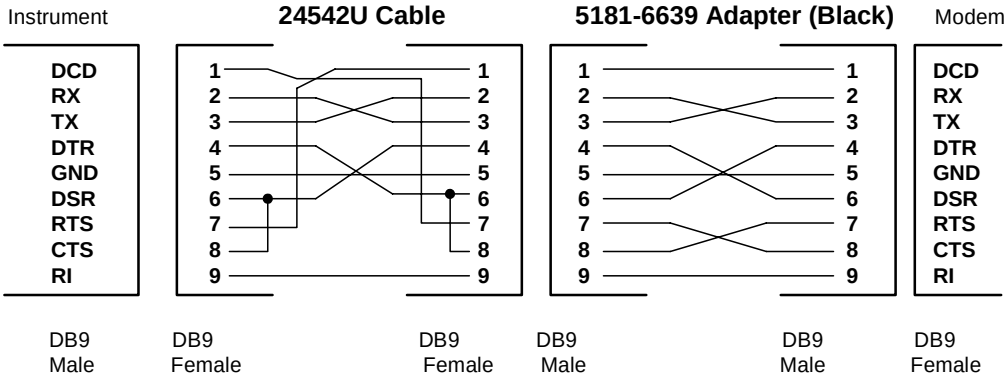
Cable/Adapter Pinouts








Instrument
24542U Cable
5181-6642 Adapter (Gray)
Modem
DCD
RX
TX
DTR
GND
DSR
RTS
CTS
RI
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 20 22
TX
RX
RTS
CTS
DSR
GND
DCD
DTR
RI
DB9 Male
DB9 Female
DB9 Female
DB9 Male
DB25 Female
DB25 Male


Instrument
24542U Cable
5181-6639 Adapter (Black)
Modem
DCD
RX
TX
DTR
GND
DSR
RTS
CTS
RI
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
DCD
RX
TX
DTR
GND
DSR
RTS
CTS
RI
DB9 Male
DB9 Female
DB9 Female
DB9 Male
DB9 Male
DB9 Female

Notes:

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 bus.

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 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. In SICL, a logical unit 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 device 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). Multi-threaded applications are only supported with 32-bit SICL.

Index

Agilent Technologies

Part Number: E2094-90037
Printed in U.S.A. E0700