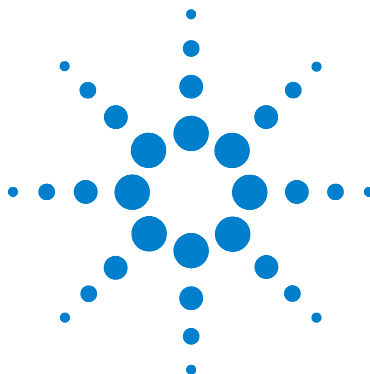


Agilent IO Libraries Suite

E2094S

Agilent SICL User’s Guide for IO Libraries Suite 15.5

Agilent Technologies

Notices

© Agilent Technologies, Inc. 1995-1996, 1998, 2000-2009

No part of this manual may be reproduced in any form or by any means (including electronic storage and retrieval or translation into a foreign language) without prior agreement and written consent from Agilent Technologies, Inc. as governed by United States and international copyright laws.

Manual Part Number

E2094-90012

Edition

First edition, October 2009

Agilent Technologies, Inc.
815 14th Street SW
Loveland, CO 80537 USA

Trademark Information

Visual Studio is a registered trademark of Microsoft Corporation in the United States and other countries.

Windows NT is a U.S. registered trademark of Microsoft Corporation.

Windows and MS Windows are U.S. registered trademarks of Microsoft Corporation.

Software Revision

This guide is valid for Revisions 15.xx of the Agilent IO Libraries Suite software, where xx refers to minor revisions of the software that do not affect the technical accuracy of this guide.

Warranty

The material contained in this document is provided "as is," and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, Agilent disclaims all warranties, either express or implied, with regard to this manual and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should Agilent and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.

Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

Restricted Rights Legend

If software is for use in the performance of a U.S. Government prime contract or subcontract, Software is delivered and licensed as "Commercial computer software" as defined in DFAR 252.227-7014 (June 1995), or as a "commercial item" as defined in FAR 2.101(a) or as "Restricted computer software" as defined in FAR 52.227-19 (June 1987) or any equivalent agency regulation or contract clause. Use, duplication or disclosure of Software is subject to Agilent Technologies' standard commercial license terms, and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 52.227-14 (June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

Safety Notices

CAUTION

A **CAUTION** notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in damage to the product or loss of important data. Do not proceed beyond a **CAUTION** notice until the indicated conditions are fully understood and met.

WARNING

A WARNING notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in personal injury or death. Do not proceed beyond a WARNING notice until the indicated conditions are fully understood and met.

Agilent SICL User's Guide for Windows

1 Introduction

What's in This Guide? 10

SICL Overview 11
- Introducing VISA, VISA COM, and SICL 11

SICL Description 12
- SICL Support 12
- SICL Users 12
- SICL Documentation 12

If You Need Help 14

2 Getting Started with SICL

Getting Started Using C 16
- C Sample Program Code 16
- C Sample Code Description 18
- Compiling the C Sample Program 19
- Running the C Sample Program 21
- Where to Go Next 21

Getting Started Using Visual Basic 22
- Visual Basic Program Sample Code 22
- Visual Basic Sample Code Description 24
- Building and Running the VB Sample Program 25
- Where to Go Next 26

3 Programming with SICL

Building a SICL Application 30
- Including the SICL Declaration File 30
- Libraries for C Applications and DLLs 30
- Compiling and Linking C Applications using Visual C++ 31

Loading and Running Visual Basic Applications 32
Thread Support for 32-bit Windows Applications 32
Opening a Communications Session 32
Sending I/O Commands 36
Handling Asynchronous Events 56
Handling Errors 59
Using Locks 64
Additional Sample Programs 68

4 Using SICL with GPIB

Introduction to GPIB Interfaces 82
GPIB Interfaces Overview 82
Typical GPIB Interface 82
Configuring GPIB Interfaces 83

Selecting a GPIB Communications Session 85
SICL GPIB Functions 85
Using GPIB Device Sessions 86

Using GPIB Interface Sessions 92
SICL Functions for GPIB Interface Sessions 92
GPIB Interface Session Code Samples 94

Using GPIB Commander Sessions 97
SICL Functions for GPIB Commander Sessions 97
Addressing GPIB Commanders 98
Writing GPIB Interrupt Handlers 98

5 Using SICL with VXI

Introduction to VXI Interfaces 104

VXI Interfaces Overview 105
Typical VXI Interface 105
Configuring VXI Interfaces 106

VXI Communications Sessions 109

VXI Device Types 109
SICL Functions for VXI Interfaces 110
Programming VXI Message-Based Devices 111
Addressing VXI Message-Based Devices 112
Programming VXI Register-Based Devices 115
Addressing VXI Register-Based Devices 116

Programming Directly to Registers 118
Mapping Memory Space for Register-Based Devices 118
Reading and Writing Device Registers 120
Sample: VXI Register-Based Programming (C) 121

Programming VXI Interface Sessions 123
VXI Interface Sessions Functions 123
Addressing VXI Interface Sessions 123

Miscellaneous VXI Interface Programming 126
Communicating with VME Devices 126
VXI Backplane Memory I/O Performance 131
Using VXI-Specific Interrupts 135

6 Using SICL with RS-232

Introduction to RS-232 Interfaces 140
ASRL (RS-232) Interface Overview 140
Configuring RS-232 (ASRL) Interfaces 141
RS-232 Communications Sessions 142
RS-232 SICL Functions 144
Using RS-232 Device Sessions 147
Using RS-232 Interface Sessions 152

7 Using SICL with LAN

Introduction to LAN Interfaces 160
Considerations when Using SICL with LAN 162
SICL LAN Functions 164

Using Remote Sessions 165
Addressing Guidelines 165
Creating a Remote Session 165
SICL Function Support 168
Remote Interface Support 168
LAN Timeout Functions 169
Sample Programs 170

Using LAN Interface Sessions 174
Addressing LAN Interface Sessions 174
SICL Function Support 174

Using Locks, Threads, and Timeouts 176
Using Locks and Threads Over LAN 176
Using Timeouts with LAN 177

8 Using SICL with USB

USB Interfaces Overview 184

Communicating with a USB Instrument Using SICL 185
Operations Supported on All USBTMC Devices 187
Operations Supported Only on USBTMC-USB488 Devices 188

A Appendix A: SICL Library Information

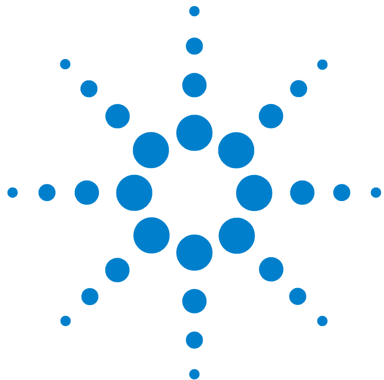
SICL Library Information 192
File System Information 192
The Registry 193

B Appendix B: Troubleshooting SICL Programs

Troubleshooting SICL Programs 196
SICL Error Codes 196
Common Windows Problems 200
Common RS-232 Problems 200

Common LAN Problems 201
General Troubleshooting Techniques 201
LAN Client Problems 202
LAN Server Problems 204

Glossary



1
Introduction

This *Agilent SICL User's Guide* describes Agilent SICL and how to use it to develop I/O applications on Microsoft Windows®. 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, VXI, RS-232, LAN, and USB interfaces.

NOTE

Before you can use SICL, you must install and configure the Agilent IO Libraries Suite on your computer. See the *Agilent IO Libraries Suite Connectivity Guide wiith Getting Started* for installation instructions.

This chapter includes:

- What's in This Guide?
- SICL Overview
- If You Need Help

What's in This Guide?

This chapter provides an introduction and overview of SICL. Subsequent chapters address the following topics:

- *Chapter 2 - Getting Started with SICL* shows how to build and run a sample program in C/C++ and in 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 VXI* shows how to communicate over the VXIbus interface.
- *Chapter 6 - Using SICL with RS-232* shows how to communicate over the RS-232 interface.
- *Chapter 7 - Using SICL with LAN* shows how to communicate over a Local Area Network (LAN).
- *Chapter 8 - Using SICL with USB* shows how to communicate over a USB interface.
- *Appendix A - SICL Library Information* provides information on SICL files and registry entries.
- *Appendix B - Troubleshooting SICL Programs* gives general troubleshooting techniques and shows common Windows, RS-232, and LAN problems.
- *Glossary* includes major terms and definitions used in this guide.

SICL Overview

SICL is part of the Agilent IO Libraries Suite product. The Agilent IO Libraries Suite includes three libraries: Agilent Virtual Instrument Software Architecture (VISA), VISA for the Common Object Model (VISA COM) and Agilent Standard Instrument Control Library (SICL).

Introducing VISA, VISA COM, 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.
- If you are using new instruments or are developing new I/O applications or instrument drivers, we recommend you use Agilent VISA or VISA COM. In particular, use VISA or VISA COM 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.
- Agilent Standard Instrument Control Library (SICL) is an I/O library developed by 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 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 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 very few, if any, 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

This 32-bit version of SICL is supported on Windows 2000, Windows XP, and Windows Vista. (For information on 16-bit SICL support, and support of older operating systems, see the revision history information in the *Agilent IO Libraries Suite Online Help*.) C, C++, and Visual Basic are supported on these Windows versions.

SICL Users

SICL is intended for instrument I/O and is best for C/C++ or Visual Basic programmers who are familiar with Windows programming. To perform SICL installation and configuration on Windows 2000, Windows XP, or Windows Vista, you must have system administrator privileges on the applicable system.

SICL Documentation

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

Table 1 Agilent SICL Documentation

Document	Description
Agilent SICL User's Guide for Windows	Shows how to use Agilent SICL.
SICL Online Help	Function reference information is provided in the form of Windows Help.
SICL Sample Programs	Sample programs are provided online to help you develop SICL applications. If the default installation directory was used, SICL sample programs are provided in the `C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples` subdirectory.
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

If You Need Help

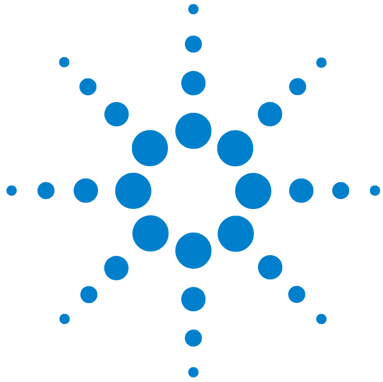
- In the USA, you can reach Agilent Technologies at this telephone number:

  USA: 1-800-829-4444
- Outside the USA, 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
- The Agilent Developer Network (ADN),

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

  is a one-stop Web resource that supports your connectivity needs with software downloads, sample code, technical notes and white papers.



2
Getting Started with SICL

This chapter provides guidelines to help you get started programming with SICL using the C/C++ and Visual Basic languages. This chapter provides sample 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

You may want to review the SICL Language Reference in online Help to familiarize yourself with SICL functions. To see the reference information online, click the IO Control (blue **IO** icon) in the Windows notification area.

Getting Started Using C

This section describes a sample program called **idn** that queries a GPIB instrument for its identification string. This sample builds a console application for WIN32 programs (32-bit SICL programs on Windows systems) using the C programming language.

C Sample Program Code

All files used to develop SICL applications in C or C++ are located in the `C` subdirectory of the base Agilent IO Libraries Suite installation directory. Sample C/C++ programs for SICL are located in the `C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\C\SICL` subdirectory, if Agilent IO Libraries Suite was installed in the default directory.

You must first compile the sample C/C++ programs before you can execute them. Some sample programs include makefiles or project files that you can use to build the programs.

The **idn** sample files are located in the `ProgrammingSamples\C\SICL\idn` subdirectory under the base Agilent IO Libraries Suite installation 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 sample 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. Edit the DEVICE_ADDRESS
line to specify the address of the applicable
device. For example:
gpib0,0: refers to a GPIB device at bus address
0 connected to an interface named ‘gpib0’ by the
Connection Expert utility.
```

```
gpib0,9,0: refers to a GPIB device at bus
address 9, secondary address 0, connected to an
interface named "gpib0" by the Connection Expert
utility. */

#include <stdio.h>/* for printf() */
#include "sicl.h"/* SICL routines */
#define DEVICE_ADDRESS "gpib0,0" /* Modify for
  setup */

void main(void)
{

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

/* Install a default SICL error handler that
logs an error message and exits. View messages
with 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);

}
```

C Sample 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 you wish, you can specify a custom error handling routine instead.

NOTE You can view SICL error messages with the **Event Viewer** utility available from the Agilent IO Control on the Windows taskbar.

iopen

When an **iopen** call is made, the parameter string *"gpib0,0"* passed to **iopen** specifies the GPIB interface followed by the bus address of the instrument. The interface name **gpib0** is the name given to the interface during execution of the Connection Expert 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 the instrument.

iclose

The **iclose** function closes the device session to this instrument (*id* is no longer valid after this point).

Compiling the C Sample Program

The `ProgrammingSamples\C\SICL\idn` subdirectory (default path `C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\C\SICL\`

idn) contains the `idn.c` source file for this sample program. Steps required to compile the **idn** sample program in Microsoft Visual C++ 6.0 follow:

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

2 In Visual C++, select **File > New...** to create a new project. Select **Win32 Console Application** for this sample program, and type in a name for your project.

3 Select **Project > Settings** from the menu. Click the **Link** tab and add `sicl32.lib` to the **Object/Library Modules** list box. Optionally, you may add the library directly to your project file. Click **OK** to close the dialog box.

4 You may want to add the *include files* and *library files* search paths. They are set as follows:

   - Select **Tools > Options** from the menu.
   - Click the **Directories** tab to set the include file path.
   - Select **Include Files** from the **Show Directories For** list box.
   - Click at the bottom of the list box and type:
     `C:\Program Files\Agilent\IO Libraries Suite \include`
     (This assumes that you used the default installation location for IO Libraries Suite.)
   - Select **Library Files** from the **Show Directories For** list box.
   - Click at the bottom of the list box and type:
     `C:\Program Files\Agilent\IO Libraries Suite \lib`
     (This assumes that you used the default installation location for IO Libraries Suite.)

5 Add or create your C or C++ source files. For this sample program, select **Project > Add to Project > Files...** and type or browse to `C:\Program Files\Agilent\ IO Libraries Suite\ProgrammingSamples\C\SICL\ idn\idn.c` (assuming the default installation location).

6 The program assumes the GPIB interface name is **gpib0** (set using the Connection Expert utility) 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.

7 Click **Build > Rebuild All** to build the SICL program.

Running the C Sample Program

- To run the **idn** sample program, execute the program from a console command prompt by selecting **Project > Execute** or **Run > Go**.

If the program runs correctly, a sample of the output if connected to a 54622A oscilloscope is:

AGILENT TECHNOLOGIES,54622A,22457869,A.01.50

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

Where to Go Next

Go to Chapter 3, "Programming with SICL." In addition, 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 VXI
- Chapter 6 - Using SICL with RS-232
- Chapter 7 - Using SICL with LAN
- Chapter 8 - Using SICL with USB

You may also want to familiarize yourself with SICL functions, which are defined in the reference information provided in the SICL online Help. If you have any problems, see "Appendix B: Troubleshooting SICL Programs" for more information.

Getting Started Using Visual Basic

This section provides guidelines to getting started programming applications in Visual Basic 6.0 (VB 6.0).

Visual Basic Program Sample Code

This section describes a sample program called **idn** that queries a GPIB instrument for its identification string. This sample builds a console application using the Microsoft Visual Basic 6.0 programming environment.

NOTE Be sure to include the `sicl32.bas` file 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.

The default Agilent IO Libraries Suite install location is `C:\Program Files\Agilent\IO Libraries Suite`. Sample Visual Basic programs for SICL are located in the `C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\VB6\SICL` subdirectory. Each sample program subdirectory contains a project file (`.vbp`) that you can open from Visual Basic 6.0.

The **idn** sample files are located in the `ProgrammingSamples\VB6\SICL\idn` subdirectory under the base Agilent IO Libraries Suite installation directory. This subdirectory contains the Visual Basic module, `idn.bas`. This module is listed on the following pages (some comments are not listed). An explanation of the function calls in the sample follows the program listing.

```
Option Explicit
''''''''''''''''''''''''''''''''''''''''''''''''
'   idn.bas
'   The following subroutine queries *IDN? on a
'   GPIB instrument and prints out the result. No
'   SICL error handling is set up in this
```

```
   example, but should be as good programming
   practice
''''''''''''''''''''''''''''''''''''''''''''''''

Sub Main()
  Dim id As Integer
  Dim strres As String * 80 ` Fixed-length
                            ` String
  Dim actual As Long
  ' Open the instrument session
  `"gpib0" is the SICL Interface name as
  ` defined in the Connection Expert
  ' "22" is the instrument gpib address on the
  ' bus
  ' Change these to the SICL Name and gpib
  ` address for your instrument

  id = iopen("gpib0,22")
  Call itimeout(id, 5000)

  ' Query device's *IDN? string
  Call iwrite(id, "*IDN?" + Chr$(10), 6, 1, 0&)

  ' Read result
  Call iread(id, strres, 80, 0&, actual)

  ' Display the results
  MsgBox "Result is: " + strres, vbOKOnly,
    "*IDN? Result"

  ' Close the instrument session
  Call iclose(id)

End Sub
```

Visual Basic Sample Code Description

id

Notice the declaration of *id* at the beginning of Sub Main(). The integer *id* 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.

iopen

When an **iopen** call is made, the parameter string *"gpib0,22"* passed to **iopen** specifies the GPIB interface followed by the bus address of the instrument. The interface name *"gpib0"* is the name given to the interface during execution of the Connection Expert utility. The bus (primary) address of the instrument follows ("*22*" 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.

NOTE You can view error messages by running the **Event Viewer** available from the Agilent IO Control in the taskbar notification area.

---

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.

iwrite and iread

The SICL I/O **iwrite** function sends a block of data to an interface or device and **iread** reads raw data from the device or interface. The **iwrite** call sends the Standard Commands for Programmable Instruments (SCPI) command ***IDN?** to the instrument that asks for its identification string.

The fixed-length string *strres* is read back into *buf* with **iread** and this is then displayed in a Message Box. SICL automatically handles all addressing and GPIB bus management necessary to perform these reads and writes to the instrument.

iclose

The **iclose** function closes the device session to this instrument (*id* is no longer valid after this point).

Building and Running the VB Sample Program

The `ProgrammingSamples\VB6\SICL\idn` subdirectory contains the files you can use to build and run the sample:

`idn.bas`	Microsoft Visual Basic 6.0 Module file
`idn.vbp`	Microsoft Visual Basic 6.0 Project file
`idn.vbw`	Microsoft Visual Basic 6.0 Workspace file

The steps to build and run the **idn** sample program follow.

1. Connect an instrument to a GPIB interface that is compatible with IEEE 488.2.
2. Start the Visual Basic 6.0 application.

NOTE This example assumes you are building a new project (no `.vbp` file exists for the project). If you do not want to build the project from scratch, from the menu select **File > Open Project...**, select and open the `idn.vbp` file, and skip to step 7.

3. Start a new Visual Basic (VB 6.0) Standard EXE project. VB 6.0 will open up a new **Project1** project with a blank Form, **Form1**.

4 From the menu, select **Project > Add Module**, select the **Existing** tab, and browse to the `idn` directory. If you used default installation paths, this directory is `C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\VB6\SICL\idn.` Select the file `idn.bas` and click **Open**.

5 (Optional) Since the **Main( )** subroutine is executed when the program is run without requiring user interaction with a Form, you may choose to delete **Form1**. To do this, right-click **Form1** in the Project Explorer window and select **Remove Form1**.

6 SICL applications in Visual Basic require that the SICL Visual Basic declaration file `sicl32.bas` module be added to your VB project. This file contains the SICL function definitions and constant declarations needed to make SICL calls from Visual Basic. To add this module to your project, from the menu select **Project > Add Module**, select the **Existing** tab, browse to the `include` directory under the Agilent IO Libraries Suite install directory (by default, this is `C:\Program Files\Agilent\IO Libraries Suite\include`), select `sicl32.bas`, and click **Open**.

7 At this point, you can run and debug the Visual Basic project.

8 The program assumes the SICL interface ID is **gpib0** (set using the Connection Expert utility) and the instrument is at bus address **22**. If necessary, modify the interface name and instrument address.

9 If the program runs correctly, an example of the output if connected to an Agilent 34401A multimeter would be:

AGILENT TECHNOLOGIES`,34401A,123456789,A.01.01`

10 If you want to make an executable file, from the menu select **File > Make idn.exe...** and click **Open**. This will create `idn.exe` in the `idn` directory.

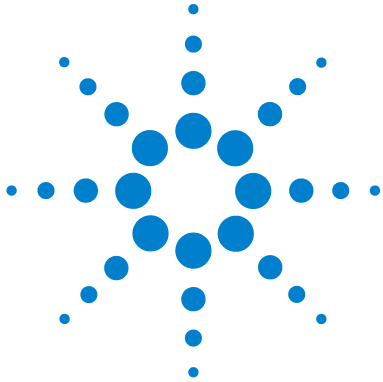
11 If the program does not run, see the message logger for a list of run-time errors and see *"Appendix B: Troubleshooting SICL Programs"* for guidelines to correct the problem.

Where to Go Next

Go to *Chapter 3*, "Programming with SICL." In addition, 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 VXI*
- *Chapter 6 - Using SICL with RS-232*
- *Chapter 7 - Using SICL with LAN*
- *Chapter 8 - Using SICL with USB*

You may also want to familiarize yourself with SICL functions, which are defined in the reference information provided in SICL online Help. If you have any problems, see *"Appendix B: Troubleshooting SICL Programs"* for more information.



3
Programming with SICL

This chapter describes how to build a SICL application and discusses SICL programming techniques. Sample programs are provided to help you develop SICL applications.

The sample programs in this chapter can be found in the following locations, if Agilent IO Libraries Suite were installed in the default directory:

For C/C++: `C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\C\SICL`

For Visual Basic:
`C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\VB6\SICL`

The chapter includes:

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

NOTE For details about SICL functions, see the *SICL Online Help*.

Building a SICL Application

This section provides guidelines to building 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 4.0 or later) programs, you must add the `sicl32.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.

The SICL libraries are located in the `lib` directory under the Agilent IO Libraries Suite base directory (for example, `C:\Program Files\Agilent\IO Libraries Suite\lib`, if you installed Agilent IO Libraries Suite 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 using Visual C++

A summary of important compiler-specific considerations for Microsoft Visual C++ follows.

NOTE

If you are using a version of Microsoft Visual Studio® other than Version 6.0, or if you are using another compiler, the menu structure and selections may be different than indicated here.

---

1. Select **Project > Settings** (or **Build > Settings** for some older Visual Studio versions) from the menu.
2. Click the **C/C++** tab. 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** tab. 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 IO Libraries Suite directories (for example, `C:\Program Files\Agilent\ IO Libraries Suite\include` and `C:\Program Files\Agilent\IO Libraries Suite\lib`) to the include file and library file search paths. To do this, select **Tools > Options** from the menu and click the **Directories** tab. Then:

   a. To set the include file path, select **Include Files** from the **Show Directories for:** list box. Next, click below the last listed path to add a new path, and type `C:\Program Files\Agilent\IO Libraries Suite\include`
   Then, click **OK**.

   b. To set the library file path, select **Library Files** from the **Show Directories for:** list box. Click below the last listed pat to add a new path, and type
   `C:\Program Files\Agilent\IO Libraries Suite\lib`
   Then, click **OK**.

Loading and Running Visual Basic Applications

To load and run an existing Visual Basic application, first start 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 `.vbp` file extension. After you have opened the application's project file, you can run the application by pressing **F5** or by clicking 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 standalone (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 to keep in mind:

- 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 7, "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 it 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 (in a non-controller role).

Opening 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 you have declared the variable, you can open the communication channel by using the SICL **iopen** function, as shown in the following code sample.

C sample:

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

Visual Basic sample:

```
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 to have 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 a symbolic interface name and a device-specific logical address in the *addr* parameter of the **iopen** function. The logical unit is an integer corresponding to the interface.

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

Table 2 Examples of Addressing Instruments

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.
gpib0,23	GPIB device at address 23.
gpib0,23,1	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 interface ID are set by running the Connection Expert utility from the Agilent IO Control (**IO** icon on the taskbar). See the *IO Libraries Suite Online Help* for details.

Examples: Opening a Device Session The following code samples open a device session with a GPIB device at bus address 23.

C sample:

```
INST dmm;
dmm = iopen ("gpib0,23");
```

Visual Basic sample:

```
Dim dmm As Integer
dmm = iopen ("gpib0,23")
```

Interface Sessions

An **interface session** allows direct, low-level control of the specified interface. A full set of interface-specific SICL functions exists 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 creates less-portable code.

Addressing Interface Sessions To create an interface session, specify the interface logical unit or interface ID in the *addr* parameter of the **iopen** function. The interface logical unit and interface ID are set by running the Connection Expert utility from the Agilent IO Control (**IO** icon on the taskbar). See the *IO Libraries Suite Online Help* for details.

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

Table 3 Valid SICL Addresses for Interfaces

7	Interface card at logical unit 7
gpib0	GPIB interface card.
gpib1	Second GPIB interface card.
com1	RS-232 interface card.

Samples: Opening an Interface Session These code samples open an interface session with an RS-232 interface.

C sample:

```
INST com1;
com1 = iopen ("com1");
```

Visual Basic sample:

```
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 the active controller, commander sessions can be used to talk to the computer that is the 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 ID or logical unit followed by a comma, and then the string **cmdr** in the **iopen** function. The following are valid commander addresses.

Table 4 Valid Commander Addresses

gpib0,cmdr	GPIB commander session.
7,cmdr	Commander session on interface at logical unit 7.

Samples: Creating a Commander Session These code samples create a commander session with the GPIB interface. The function calls open a session of communication with the commander on a GPIB interface.

C sample:

```
INST cmdr;
cmdr = iopen("gpib0,cmdr");
```

Visual Basic sample:

```
Dim cmdr As Integer
cmdr = iopen ("gpib0,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 or from 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, the formatting 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 follow.

- 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” on page 53.) These are raw I/O functions and should not be intermixed 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” on page 52 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]format code
```

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

Table 5 Format Flags

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 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 pad 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 is an optional integer preceded by a period. When used with format codes `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** format codes, the minimum number of digits to appear is specified. For the **`s`** and **`S`** format codes, 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`** format codes). 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 format code.

Table 6 Argument Modifiers in C Applications

Argument Modifier	Format Codes	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.

Format Codes for sending and receiving formatted I/O are different. The following tables summarize the format codes for each.

Table 7 iprintf and ipromptf Format Codes in C Applications

Format Codes	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.

Table 8 iscanf and ipromptf Format Codes

Format Codes	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);
```

Sample: Formatted I/O (C) shows one way to send and receive formatted I/O. This code sample opens a GPIB communications session with a multimeter and uses a comma operator to send a comma-separated list to the multimeter. The *lf* format codes 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};

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

/* Open the multimeter session */
dvm = iopen ("gpib0,16");
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);

  return 0;
}
```

Format Strings 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 the point at which the data is 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. You must make 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. You can change this behavior with the **isetbuf** and **isetubuf** functions. See "Formatted I/O Buffers" on page 52 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 a 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` format code 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 from 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.

Table 9 Functions Related to Formatted I/O

I/O Function	Description
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. You should also 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, which 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, which 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 of online Help.

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]format code
```

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

Table 10 Format Flags for ivprintf in Visual Basic

Format Flag	Description
@1	Converts to a 488.2 NR1 number.

Table 10 Format Flags for ivprintf in Visual Basic

Format Flag	Description
@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 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 is an optional integer preceded by a period. When used with format codes **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** format codes, 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** format codes). 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 optional argument modifier **h**, **l**, **w**, **z**, or **Z** is dependent on the format code.

Table 11 Argument Modifiers in Visual Basic Application

Argument Modifier	Format Codes	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.

Format Codes for sending and receiving formatted I/O are different. The following tables summarize the format codes for each.

Table 12 ivprintf Format Codes in Visual Basic Application

Format Codes	Description
d, i	Corresponding argument is an Integer.

Table 12 ivprintf Format Codes in Visual Basic Application

Format Codes	Description
b, B	Not supported in Visual Basic.
c,C	Not supported in Visual Basic.
t	Not supported in Visual Basic.
s,S	Not supported in 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.

Table 13 ivscanf format codes in Visual Basic Application

Format Codes	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 in 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)

Option Explicit
''''''''''''''''''''''''''''''''''''''''''''''''
'nonfmt.bas
'The following subroutine measures AC voltage
'on a multimeter and prints out the results.
''''''''''''''''''''''''''''''''''''''''''''''''

Sub Main()

  Dim dvm As Integer
  Dim strres As String * 20  'Fixed-length String
  Dim actual As Long

  'Open the multimeter session
  '"gpib0" is the SICL Interface name as defined
  'in the Connection Expert
  '"23" is the instrument gpib address on the bus
  'Change these to the SICL Name and gpib address
  'for your instrument

  dvm = iopen("gpib0,23")
  Call itimeout(dvm, 5000)

  'Initialize dvm
  Call iwrite(dvm, "*RST" + Chr$(10), 5, 1, 0&)

  'Set up multimeter and take measurements
  Call iwrite(dvm, "CALC:DBM:REF 50" + _
    Chr$(10), 16, 1, 0&)

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

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

  'Display the results
  MsgBox "Result is " + Left$(strres, actual)
```

```
   'Close the multimeter session
   Call iclose(dvm)

   Exit Sub

End Sub
```

Format Strings In the format string for **ivprintf**, when the special characters *Chr$(10)* are 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 These functions are related to formatted I/O in Visual Basic:

Table 14 Related Formatted I/O Functions

I/O Function	Description
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 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` Function 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 sample:

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

VB sample:

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

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

C sample:

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

VB sample:

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

Sample: 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;

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

  /* Open the multimeter session */
  dvm = iopen ("gpib0,16");
  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);

return 0; }
```

Sample: 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("gpib0,16")
  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)

  Exit Sub

End Sub
```

Handling Asynchronous Events

Asynchronous events are events that happen outside the control of your application. These events include service requests (**SRQs**) 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 calls **iclose**. Some example declarations are:

```
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 2000 and XP), 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 (gpib0, act_isr);
isetintr (gpib0, I_INTR_INTFACT, 1);
...
```

```
iintroff ();
igpibpassctl (gpib0, ba);
while (!done)
iwaithdlr (0);
iintron ();
...
```

Handling Errors

This section provides guidelines to handling 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** to log SICL error messages. Run the **Event Viewer** *after* you run the SICL program.

Using the Event Viewer SICL logs internal messages as Windows 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 Agilent IO Control (**IO** icon on the taskbar) and clicking **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 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. Error 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.

Table 15 Reserved Values for *proc*

I_ERROR_EXIT	This value installs a special error handler which will log a diagnostic message and then terminate the process.
I_ERROR_NOE XIT	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.

Sample: Installing 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 code sample, 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;

  ionerror (I_ERROR_EXIT);
  dvm = iopen ("gpib0,16");
  itimeout (dvm, 10000);
  iprintf (dvm, "%s\n", "MEAS:VOLT:DC?");
  iscanf (dvm, "%lf", &res);
  printf ("Result is %lf\n", res);
  iclose (dvm);

  return 0;
  }
```

Sample: 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;
```

```
ionerror (err_handler);
dvm = iopen ("gpib0,16");
itimeout (dvm, 10000);
iprintf (dvm, "%s\n", "MEAS:VOLT:DC?");
iscanf (dvm, "%lf", &res);
printf ("Result is %lf\n", res);
iclose (dvm);

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, you need not insert special error handling code 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. **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.

Sample: Error Handlers (Visual Basic) In this Visual Basic code sample, 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.

```
Option Explicit
''''''''''''''''''''''''''''''''''''''''''''''''
'errhand.bas
'In this example, the error handler displays the
'error message in a Message Box and then
'terminates the program.
''''''''''''''''''''''''''''''''''''''''''''''''

Sub Main()

  Dim dvm As Integer
  Dim res As Double

  'Install an error handler
  On Error GoTo ErrorHandler

  '"gpib0" is the SICL Interface name as
  'defined in Connection Expert
```

```
  '"22" is the instrument gpib address on the bus
  'Change these to the SICL Name and gpib address
  'for your instrument

  dvm = iopen("gpib0,22")

  'Set timeout to 5 seconds
  Call itimeout(dvm, 5000)

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

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

  MsgBox "Result is " + Format(res)

  iclose (dvm)

  'Tell SICL to cleanup for this task
  Call siclcleanup

  Exit Sub

ErrorHandler:

  'Display the error message
  MsgBox "*** Error : " + Error, vbExclamation

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 it 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 VEE applications use SICL interface sessions. Therefore, I/O operations from VEE applications can supersede 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 7*, "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. 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” on page 64, 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 is to use locking per transaction. Per transaction means that you lock before you set up the device, then unlock after all desired data has been acquired. When sharing a device, you cannot assume the state of the device, so the beginning of each transaction should have any setup needed to configure the device or devices to be used.

Sample: Device Locking (C)

```
/* 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;


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

  /* Open the multimeter session */
  dvm = iopen ("gpib0,16");
  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);

return 0;}
```

Sample: Device Locking (Visual Basic)

```
Option Explicit
''''''''''''''''''''''''''''''''''''''''''''''''
' locking.bas
' This example shows how device locking can be
  used to gain exclusive access to a device
''''''''''''''''''''''''''''''''''''''''''''''''

Sub Main()

  Dim dvm As Integer
  Dim strres As String * 20 'Fixed length String
  Dim actual As Long
```

```
  'Install an error handler
  On Error GoTo ErrorHandler

  'Open the multimeter session
  dvm = iopen("gpib0,23")
  Call itimeout(dvm, 10000)

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

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

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

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

  'Display the results
   MsgBox "Result is " + Left$(strres, actual)

  'Close the multimeter session
   Call iclose(dvm)

   Exit Sub

ErrorHandler:

  'Display the error message.
  MsgBox "*** Error : " + Error


End Sub
```

Additional Sample Programs

This section contains two additional sample programs that provide guidelines to help you develop SICL applications.

Sample: Oscilloscope Program (C)

This C sample programs an oscilloscope (such as an Agilent 54601), uploads the measurement data, and instructs the oscilloscope to print its display to a printer. This program uses many SICL features and illustrates some important C and Windows programming techniques for SICL.

Program Files The oscilloscope sample files are located in the `C:\Program Files\Agilent\IO Libraries Suite\c\samples\scope` subdirectory, if Agilent IO Libraries Suite was installed in the default directory. The subdirectory contains the source program and a number of files to help you build the sample with specific compilers, depending on the Windows environment used.

Table 16 Program Files for the C Oscilloscope Program

`SCOPE.C`	Sample program source file.
`SCOPE.H`	Sample program header file.
`SCOPE.RC`	Sample program resource file.
`SCOPE.ICO`	Sample program icon file.

Building the Project File This section shows how to create the project file for this sample using Microsoft Visual C++ 6.0.

To compile and link the sample program with Microsoft Visual C:

1. Select **File > New** from the menu. Select the **Project** tab.
2. Type the name you want for the project in the edit box labeled **Project name**. Then, select **Win32 Application** from the project type list box. Specify a directory location for the project in the **Location** edit box. Click the **OK** button.
3. The Win32 Application wizard will appear. Select **An empty project** and click **Finish**.
4. Click **Project > Add to Project > Files**.... Browse to the sample folder (by default, this is `C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\C\SICL\scope`). Select the source files `scope.c`, `scope.rc`, and

scope.h to add them to the project. Also add sicl32.lib from the lib directory (by default, C:\ Program Files\Agilent\IO Libraries Suite\lib).

5 Select **Project** > **Settings** from the menu and click the **C/C++** tab. Select **Code Generation** from the **Category** list box. Then, select **Multithreaded DLL** from the **Use run-Time library** list box and click **OK**.

6 Select **Tools** > **Options** from the menu and click the **Directories** tab in the **Options** dialog box. Select **Include Files** from the **Show Directories for:** list box, click the **New** icon, click below the last directory in the list box, browse to the IO Libraries Suite include directory (by default, C:\Program Files\Agilent\ IO Libraries Suite\include) and click **OK**.

7 Select **Build** > **Build** *samplename.exe* to build the application.

If there are no errors reported, you can execute the program by selecting **Build** > **Execute** *samplename.exe*. An application window will open. Several commands are available from the **Action** 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 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 program illustrates specific SICL features and programming techniques and is not meant to be a robust Windows application. See 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, eterrstr(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.

Formatted I/O SICL provides extensive formatted I/O functionality to help facilitate communication of I/O commands and data. The sample 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 modifier 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 sample 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 gpib0intf ;
  ...
  gpib0intf = 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 of the 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(gpib0intf,cmd,length);
  igpibatnctl(gpib0intf,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 sample 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 sample 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 sample, 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!") ;
  ...
  }
```

Sample: Oscilloscope Program (Visual Basic)

This Visual Basic sample 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 sample files are located in the `C:\Program Files\Agilent\IO Libraries Suite\vb\samples\scope` subdirectory, if Agilent IO Libraries Suite was installed in the default directory. The files are listed in the following table.

Table 17 Files Used for the Oscilloscope Sample Program

`SCOPE.FRM`	Visual Basic source for the SCOPE sample program.
`SCOPE.VBP`	Visual Basic project file for the SCOPE sample program.
`SCOPE.VBW`	Visual Basic workspace file for the SCOPE sample 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 6.0.
3. Open the project file `scope.vbp` 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. If a Project Tree is not already visible, select **View > Project Explorer** from the Visual Basic menu.
   b. Under **Forms**, right-click **scope.frm** and select **View Code**.
   c. Edit the following line so the address is set to the address of the oscilloscope:

   ```
   Private Const scope_address = "gpib0,7" '
     Address of SCOPE
   ```
5. Run the program by pressing the **F5** key or by clicking 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.
8. After performing these steps, you can create a standalone executable (`.exe`) version of this program by selecting **File > Make scope.exe...** from the Visual Basic menu.

Program Overview 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 program illustrates specific SICL

features and programming techniques and is not meant to be a robust Windows application. See the SICL online Help for detailed information on the SICL features used in this program.

Table 18 Functions of the Sample Program

Listing	Description
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 sample, 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 sample, the default address is “gpib0,7.” The interface name **gpib0** is the name given to the interface with the Connection Expert 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.

Table 18 Functions of the Sample Program

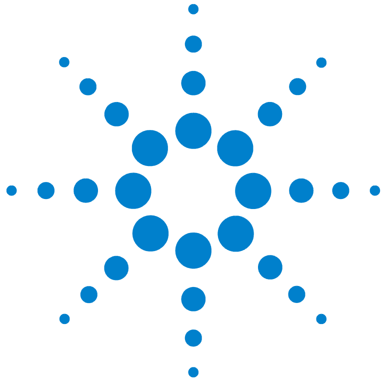
Listing	Description
igetintfsess	**igetintfsess** is called to return an interface session *id* for the interface to which the oscilloscope instrument is connected. This interface session will be used by the following **iclear** call to send an interface clear to reset the interface.
iclear	The **iclear** 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$(10)* is appended to the format string passed as the second argument to **ivprintf**. This tells **ivprintf** to flush the formatted I/O write buffer after writing the string specified in the format string.
ivscanf	The **ivscanf** function is called to read the oscilloscope's preamble information into the preamble array. The preamble array is passed as the third parameter to **ivscanf**. This passes the address of the first element of the preamble array to the **ivprintf** SICL function.
ivprintf	**ivprintf** is called to prompt the oscilloscope for its waveform data. Again, *Chr$(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.

Table 18 Functions of the Sample Program

Listing	Description
iread	**iread** is called to read in the oscilloscope's waveform. The waveform is read in as a specified number of bytes. The format string passed as the third parameter to **iread** specifies that a maximum of 2010 Byte values be read into the Byte array. A null value, *vbNull*, is passed as the fourth value and a Long variable, *actual*, returns the number of bytes actually read. ***0&*** may also be used for a null value.
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. The error returned is the most recent run-time error when no argument is passed to the function.
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**.

Table 18 Functions of the Sample Program

Listing	Description
CmdGetWaveform.Enable d	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.



4
Using SICL with GPIB

This chapter shows how to open a communications session and communicate with GPIB devices, interfaces, or controllers. The sample programs in this chapter can be found in the following locations, if Agilent IO Libraries Suite was installed in the default directory:

For C/C++:

```
C:\Program Files\Agilent\IO Libraries Suite\
ProgrammingSamples\C\SICL\
```

For Visual Basic:

```
C:\Program Files\Agilent\IO Libraries Suite\
ProgrammingSamples\VB6\SICL\
```

This chapter includes:

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

Introduction to GPIB Interfaces

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

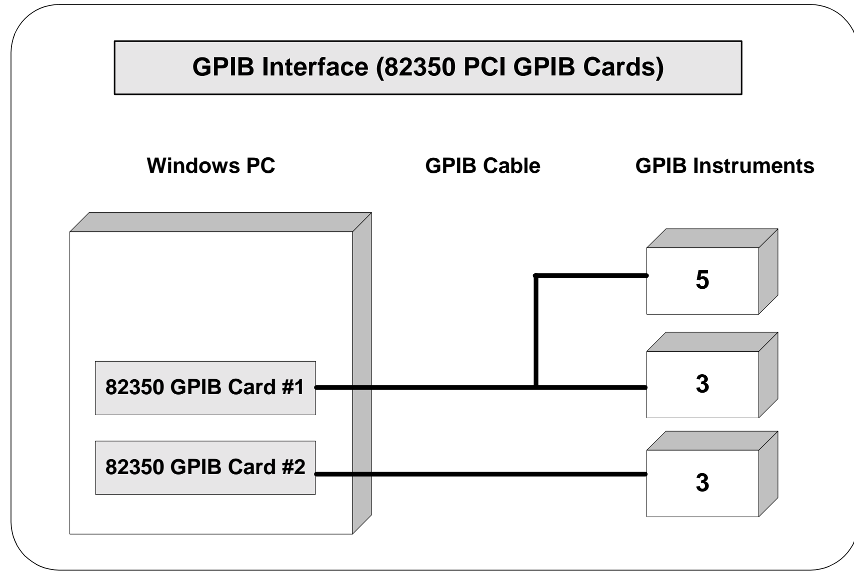
- GPIB Interfaces Overview
- Selecting a GPIB Communications Session
- SICL GPIB Functions

GPIB Interfaces Overview

This section provides an overview of GPIB interfaces, including typical hardware configuration using the Connection Expert utility, and example configurations using SICL.

Typical GPIB Interface

As shown in the following figure, a typical GPIB interface consists of a Windows PC with one or more GPIB cards (PCI and/or ISA) cards installed in the PC and one or more GPIB instruments connected to the GPIB cards via GPIB cable. I/O communication between the PC and the instruments is via the GPIB cards and the GPIB cable. This figure shows GPIB instruments at addresses 3 and 5.



Configuring GPIB Interfaces

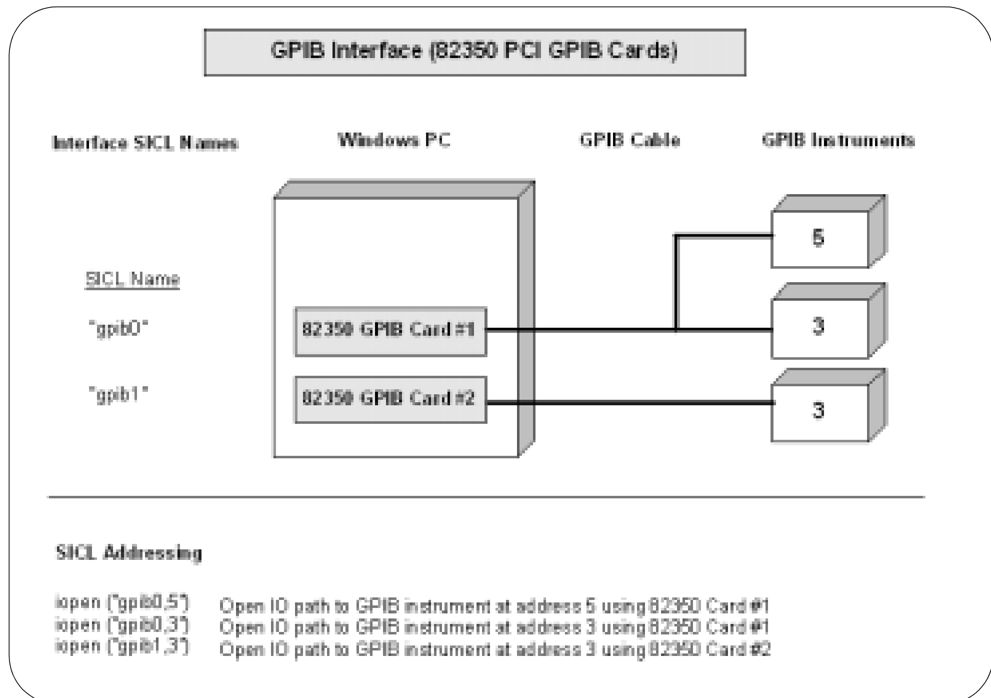
An **IO interface** can be defined as both a hardware interface and as a software interface. One function of the Connection Expert utility is to associate a unique interface name with a hardware interface.

SICL uses an **Interface Name** or **Logical Unit Number** to identify an interface. This information is passed in the parameter string of the **iopen** function call in a SICL program. Connection Expert assigns a default Interface Name and Logical Unit Number, as well as other necessary configuration values, when the interface hardware is configured; you can change these values by running the Connection Expert utility. See the *IO Libraries Suite Online Help* for details.

Example: GPIB (82350) Interface

The GPIB interface system in the following figure consists of a Windows PC with two 82350 GPIB cards connected to three GPIB instruments via GPIB cables. For this system, the Connection Expert

utility has been used to assign GPIB card #1 a SICL name of **gpib0** and to assign GPIB card #2 a SICL name of **gpib1**. With these names assigned to the interfaces, the SICL addressing is as shown in the figure. Since unique names have been assigned by Connection Expert, you can use the **iopen** command to open the I/O paths shown.



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” in this chapter.

SICL GPIB Functions

Table 19 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.

SICL Functions for GPIB Device Sessions

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.

Table 20 SICL Functions for GPIB Sessions

Function	Description
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.

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 Connection Expert utility.

Opening Connection Expert To open the Connection Expert utility, click the Agilent IO Control (**IO** icon on the taskbar) and click **Agilent Connection Expert**. See the *IO Libraries Suite Online Help* for details on this 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:

Table 21 GPIB Primary and Secondary Addresses

GPIB,7	A device address corresponding to the device at primary address 7.
gpib0,3, 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 Code Samples” for a sample program to communicate with a VXI mainframe via the GPIB interface.

Sample code to open a device session with a GPIB device at bus address 16 follows.

C sample:

```
INST dmm;
dmm = iopen (“gpib0,16”);
```

Visual Basic sample:

```
Dim dmm As Integer
dmm = iopen (“gpib0,16”)
```

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.

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.

GPIB Device Session Code Samples

This section provides C language and Visual Basic language sample programs for GPIB device sessions.

Sample: GPIB Device Session (C) This sample 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.

```
/* gpibdev.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;

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

```
/* Open the multimeter and switch sessions*/
dvm = iopen ("gpib0,9,3");
sw = iopen ("gpib0,9,14");
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);

return 0;
}
```

Sample: GPIB Device Session (Visual Basic) This sample 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.

```
Option Explicit
''''''''''''''''''''''''''''''''''''''''''''''''
' gpibdv.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
  '"gpib0" is the SICL Interface name as defined
  'in Connection Expert

  'Change this to the SICL name you have defined

  dvm = iopen("gpib0,9,3")
  sw = iopen("gpib0,9,14")

  ' set timeouts
  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
  'default form, (Name) "Form1", containing no
  ` controls)

  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 switch
      argcount = ivprintf(sw, "TRIG" + Chr$(10))
    Next i

    ' Close the sessions
    Call iclose(dvm)
    Call iclose(sw)

    ' Tell SICL to cleanup 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.

SICL Functions for GPIB Interface Sessions

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

Table 22 Implementing SICL Functions for GPIB

Function	Description
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.

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 Connection Expert utility.

Opening Connection Expert To open the Connection Expert utility, click the Agilent IO Control (**IO** icon on the taskbar) and click **Agilent Connection Expert**. See the *IO Libraries Suite Online Help* for details on this utility. Example interface addresses follow.

Table 23 Interface Names

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 code samples open an interface session with the GPIB interface.

C sample:

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

Visual Basic sample:

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

GPIB Interface Sessions Interrupts There are specific interface session interrupts that can be used. 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. For more information, see “Writing GPIB Interrupt Handlers” in this chapter.

GPIB Interface Session Code Samples

This section provides C language and Visual Basic language sample programs for GPIB interface sessions.

Sample: GPIB Interface Session (C)

```
/* gpibstat.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 */


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

  /* open GPIB interface session */
  id = iopen(“gpib0”);

  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;
}
```

Sample: GPIB Interface Session (Visual Basic)

```
'gpibstat.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(“gpib0”)
  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 the active controller.

NOTE

Because the Agilent 82357 USB/GPIB Interface Converter and the Agilent E5810 LAN to GPIB Gateway do not support non-controller roles, they also do not support GPIB commander sessions.

SICL Functions for GPIB Commander Sessions

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

Table 24 SICL Functions for GPIB Commander Sessions

Function	Description
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.

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 Connection Expert utility. To open Connection Expert, click the Agilent IO Control (**IO** icon on the taskbar) and click **Agilent Connection Expert**. See the *IO Libraries Suite Online Help* for details on this utility. Example GPIB addresses for commander sessions follow.

Table 25 Addressing GPIB Commanders

GPIB,cmdr	A commander session with the GPIB interface.
gpib0,cmdr	A commander session with the gpib0 interface.
7,cmdr	A commander session with the interface at logical unit 7.

These code samples open a commander session with the GPIB interface.

C sample:

```
INST gpib;
gpib = iopen ("gpib0,cmdr");
```

Visual Basic sample:

```
Dim gpib As Integer
gpib = iopen ("gpib0,cmdr")
```

GPIB Commander Sessions Interrupts There are specific commander session interrupts that can be used. See “Writing GPIB Interrupt Handlers” in the following section 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.

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.

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
}
```

Sample: Servicing Requests (C) This sample 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.

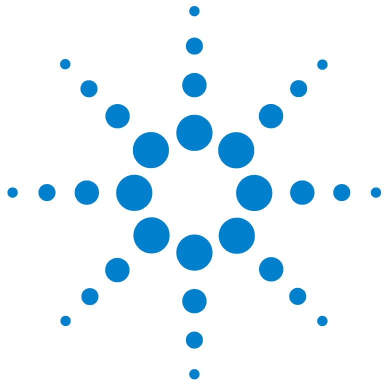
Since the program cannot leave the handler until all devices have released SRQ, it is recommended that the handler do as little as possible for each device. The previous sample assumed that only one **iscanf** was needed to service the SRQ. If lengthy operations are needed, a better technique is to perform 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 that 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 the
          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(“gpib0, 17”);
    id2 = iopen(“gpib0, 18”);

    /* Interface session for SRQ test */
    bus = iopen(“gpib0”);

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



5
Using SICL with VXI

This chapter shows how to use SICL to communicate over the VXIbus. The sample programs in this chapter can be found in the following locations, if Agilent IO Libraries Suite was installed in the default directory:

For C/C++: `C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\C\SICL\`

For Visual Basic:
`C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\VB6\SICL\`

This chapter includes:

- Introduction to VXI Interfaces
- Programming VXI Message-Based Devices
- Programming VXI Register-Based Devices
- Programming VXI Interface Sessions
- Miscellaneous VXI Interface Programming

Introduction to VXI Interfaces

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

- VXI Interfaces Overview
- VXI Communications Sessions
- VXI Device Types
- SICL Functions for VXI

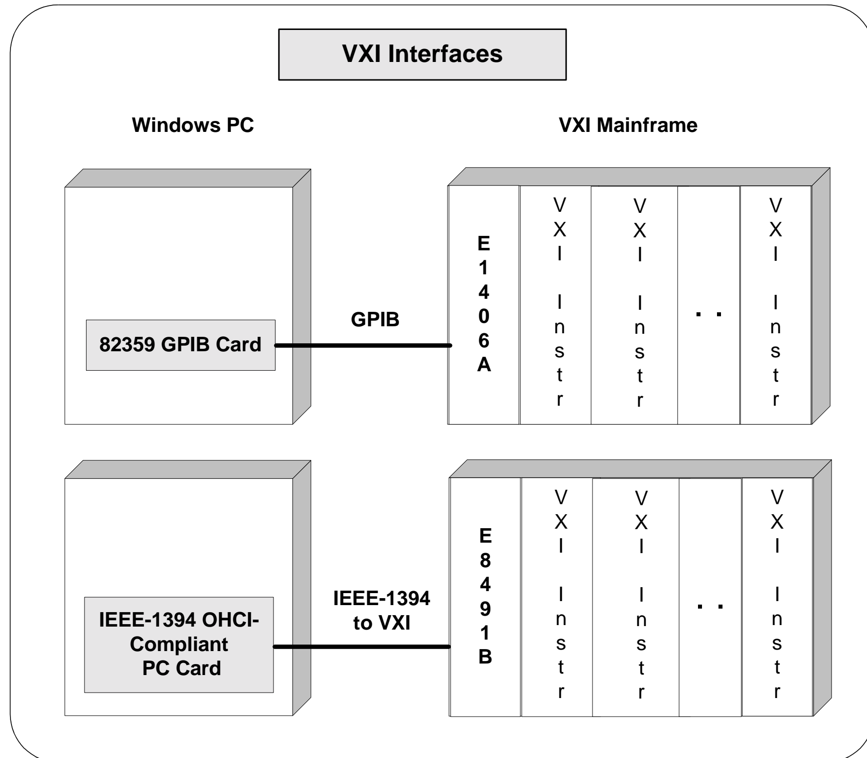
VXI Interfaces Overview

This section provides an overview of VXI interfaces, including typical hardware configuration, using Connection Expert, and example configuration using SICL.

Typical VXI Interface

As shown in the following figure, a typical VXI interface consists of one of two main hardware configurations: E1406A Command Module or E8491B IEEE-1394 to VXI Module.

- The **E1406A Command Module** version consists of a Windows PC with an 82350 (or equivalent) GPIB card and a VXI mainframe with an E1406A Command Module and one or more VXI instruments. I/O communication from the PC to the VXI instruments is via the GPIB card, GPIB cable, and E1406A Command Module.
- The **E8491B Module** version consists of a Windows PC with an IEEE-1394 OHCI-Compliant (FireWire) PC card and a VXI mainframe with an E8491B IEEE-1394 to VXI Module and one or more VXI instruments. I/O communication from the PC to the VXI instruments is via the PC card, IEEE-1394 to VXI cable, and E8491B Module.



Configuring VXI Interfaces

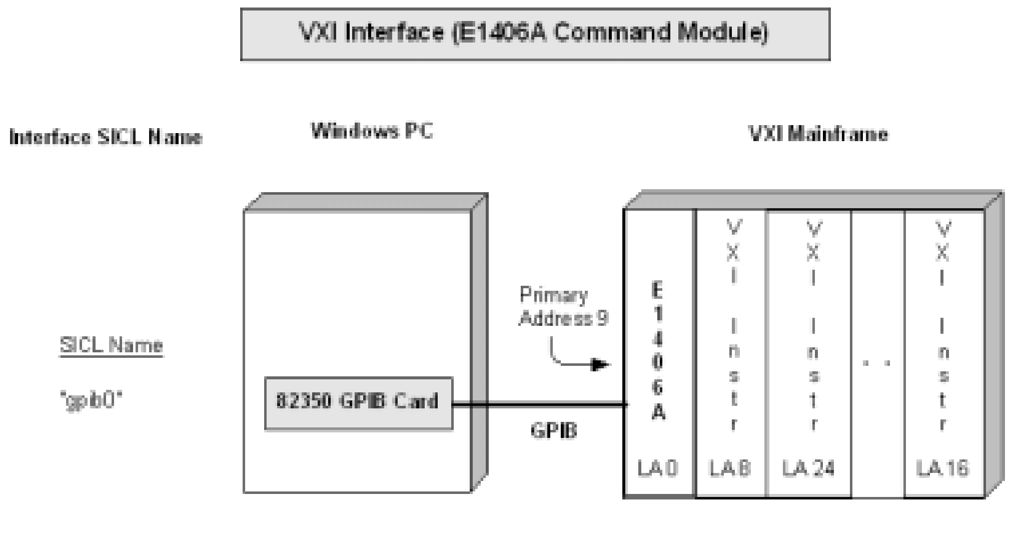
An **IO interface** can be defined as both a hardware interface and as a software interface. One function of the Connection Expert utility is to associate a unique interface name with a hardware interface.

SICL uses an **Interface Name** or **Logical Unit Number** to identify an interface. This information is passed in the parameter string of the **iopen** function call in a SICL program. Connection Expert assigns a default Interface Name and Logical Unit Number, as well as other necessary configuration values, when the interface hardware is configured; you can change these values by running the Connection Expert utility. See the *Connect IO Works Online Help* for details.

Example: VXI (E1406A) Interface

The VXI interface system in the following figure consists of a Windows PC with an 82350 GPIB card that connects to an E1406A Command Module in a VXI Mainframe. The VXI mainframe includes one or more VXI instruments. The E1406A is configured for primary address 9 and logical address (LA) 0. The three VXI instruments shown have logical addresses 8, 16, and 24.

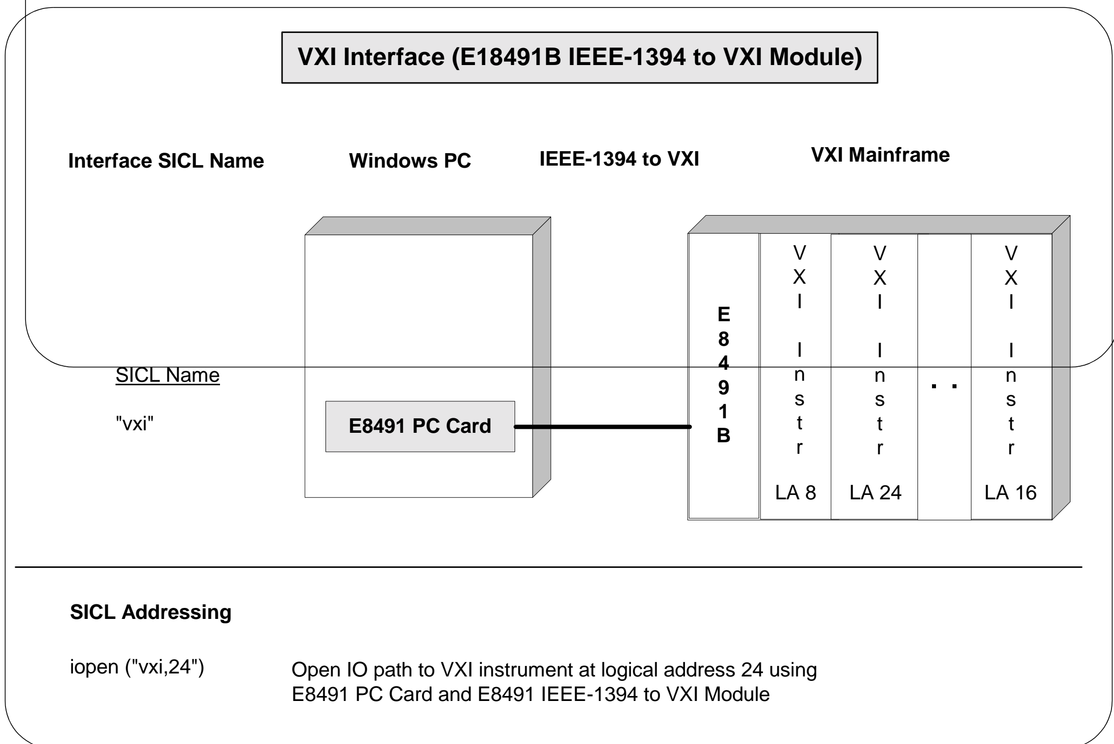
The Connection Expert utility has been used to assign the 82350 GPIB card a SICL name of **gpib0**. With these names assigned to the interfaces, the SICL addressing is as shown in the figure. For information on the E1406A Command Module, see the *Agilent E1406A Command Module User's Guide*. For information on VXI instruments, see the applicable *VXI Instrument User's Guide*.



Example: VXI (E8491) Interface

The VXI interface system in the following figure consists of a Windows PC with an E8491 PC card that connects to an E8491B IEEE-1394 to VXI Module in a VXI Mainframe. The VXI mainframe includes one or more VXI instruments. For this system, the three VXI instruments shown have logical addresses 8, 16, and 24.

The Connection Expert utility has been used to assign the E8491 PC card a SICL name of **vxi**. With this name assigned to the interface, you can use the SICL addressing shown in the figure. For information on the E8491B module, see the *Agilent E8491B User's Guide*. For information on VXI instruments, see the applicable *VXI Instrument User's Guide*.



VXI Communications Sessions

Before you begin programming your VXI system, ensure that 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 follow. Commander Sessions are *not* supported with VXI interfaces.

- **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.

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 Standard Commands for Programmable Instruments (SCPI) commands. When using SICL, place the SCPI command within the SICL output function call and the message-based device then 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. Interpreted SCPI (I-SCPI) interprets high-level SCPI commands and sends the data to the instrument. 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.
- **Direct 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.

SICL Functions for VXI Interfaces

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.

Table 26 SICL Functions for VXI Interfaces

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

Programming 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 "Programming VXI Register-Based Devices" for information on register programming.

VXI Message-Based Device Functions

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

Table 27 VXI Device Functions

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

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 Connection Expert utility. To open Connection Expert, click the Agilent IO Control (**IO** icon on the taskbar) and click **Agilent Connection Expert**. See the *IO Libraries Suite Online Help* for details on this utility.

Addressing Guidelines

Primary address must be between 0 and 255. The primary address corresponds to the VXI logical address and specifies the address in the 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.

Table 28 Addressing VXI Instruments

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”);
```

Sample: VXI Message-Based Device Session (C)

This sample 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 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 (“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);

}
```

Sample: VXI Message-Based Device Session (Visual Basic)

```
''''''''''''''''''''''''''''''''''''''''''''''''
' vximdev.bas
' This example program opens a communication
' session with a VXI message-based device and
' measures the DC voltage. Then measurement
' results are printed.
''''''''''''''''''''''''''''''''''''''''''''''''

Sub Main()

  Dim id As Integer
  Dim strres As String * 80  'Fixed-length String
  Dim actual As Long

  ' Open the instrument session

  ' "vxi" is the SICL Interface name as defined
  ' in Connection Expert
  '"216" is the instrument logical address.
  'Change these to the SICL name and logical
  ' address for your instrument

  id = iopen("vxi,216")
```

```
  '  Set timeout to 10 seconds
  Call itimeout(id, 10000)

  ' Initialize dvm
  Call iwrite(id, "*RST" + Chr$(10), 6, 1, 0&)

  ' Take measurement
  Call iwrite(id, "MEAS:VOLT:DC? 1, 0.001" + _
    Chr$(10), 23, 1, 0&)

  ' Read result
  Call iread(id, strres, 80, 0&, actual)

  ' Display the results
  MsgBox "Result is: " + strres, vbOKOnly, _
    "DVM DCV Result"

  ' Close the instrument session
  Call iclose(id)

  ' Tell SICL to clean up for this task
  Call siclcleanup

End Sub
```

Programming VXI Register-Based Devices

You can use one or more of the following methods to communicate with VXI register-based devices.

- **I-SCPI Interface Programming**. 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. Agilent VISA must be installed to use the **iscpi** interface.
- **Direct 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 Programming**. The Compiled SCPI (C-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. See the applicable C-SCPI documentation for programming information.
- **Command Module Programming**. 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 Connection Expert utility. To open Connection Expert, click the Agilent IO Control (**IO** icon on the taskbar) and click **Agilent Connection Expert**. See the *IO Libraries Suite Online Help* for details on this utility.

Functions Not Supported

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.

Table 29 Unsupported Functions

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

Table 29 Unsupported Functions

Category	Functions Not Supported
Service Requests	igetonsrq, ionsrq
Timeouts	igettimeout, itimeout
VXI Specific	ivxiws

Addressing Guidelines

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.

Table 30 Addressing Guidelines

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");
```

Programming Directly to 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, number of pages, and a suggested starting location respectively. *map_space* determines the memory location to map the space to.

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.

Table 31 Mapping Memory Space

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 device's A24 or A32 extended memory */
ptr=imap (id, I_MAP_EXTEND, 0, 1, 0);
```

```
/* Map to 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.

Table 32 Mapping Memory Space

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 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. An example using **iwpeek** follows.

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

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.

Sample: VXI Register-Based Programming (C)

This sample 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 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 communication 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);}
```

Programming 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.

VXI Interface Sessions Functions

The following table 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**).

Table 33 Implementing SICL Function for VXI

Function Name	Action
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.

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 Connection Expert utility. To open Connection Expert, click the Agilent IO Control (**IO** icon on the taskbar) and click **Agilent Connection Expert**. See the *IO Libraries Suite Online Help* for details on this utility.

Addressing Guidelines

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.

Table 34 Symbolic Interface Names

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

This example opens an interface session with the VXI interface.

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

Sample: VXI Interface Session (C)

This sample 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);
}
```

Miscellaneous VXI Interface Programming

This section provides other information for programming via the VXI interface, including:

- Communicating with VME Devices
- VXI Backplane Memory I/O Performance
- Using VXI-Specific Interrupts

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 processes when using VME devices in a VXI mainframe:

- Declaring Resources
- Mapping VME Memory
- Reading and Writing 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 the 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 Agilent controllers.

Table 35 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 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. 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.

Sample: VME Interrupts (C)

This ANSI C sample 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 is 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 sample 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);
}
```

VXI Backplane Memory I/O Performance

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

Table 36 VXI Supported Memory I/O Mechanisms

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.

Sample: VXI Memory I/O (C)

A code sample 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

Sample: 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.

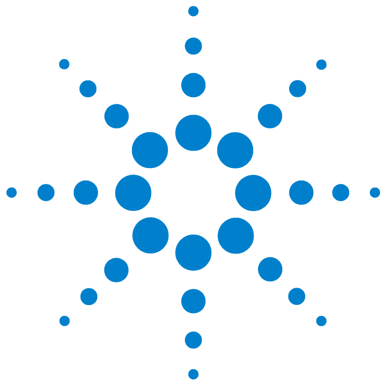
```
VME Interrupt arrives:
get iack value

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
```

Sample: 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;
 }
```



6
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 sample programs in this chapter can be found in the following locations, if Agilent IO Libraries Suite was installed in the default directory:

For C/C++: `C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\C\SICL`

For Visual Basic:
`C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\VB6\SICL`

The chapter includes:

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

Introduction to RS-232 Interfaces

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

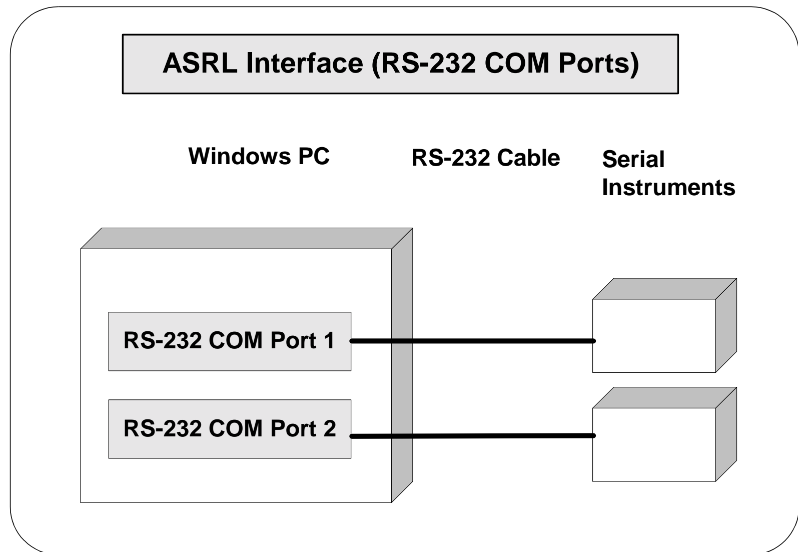
- ASRL (RS-232) Interface Overview
- Configuring RS-232 Interfaces
- RS-232 Communications Sessions
- RS-232 SICL Functions

ASRL (RS-232) Interface Overview

This section provides an overview of RS-232 interfaces, including typical hardware configuration, using the Connection Expert utility, and example configuration using SICL.

Typical RS-232 Interface

As shown in the following figure, a typical ASRL (RS-232) interface consists of a Windows PC with one or more RS-232 COM Ports. Each COM port can be connected to one, and only one, Serial instrument via an RS-232 cable.



Configuring RS-232 (ASRL) Interfaces

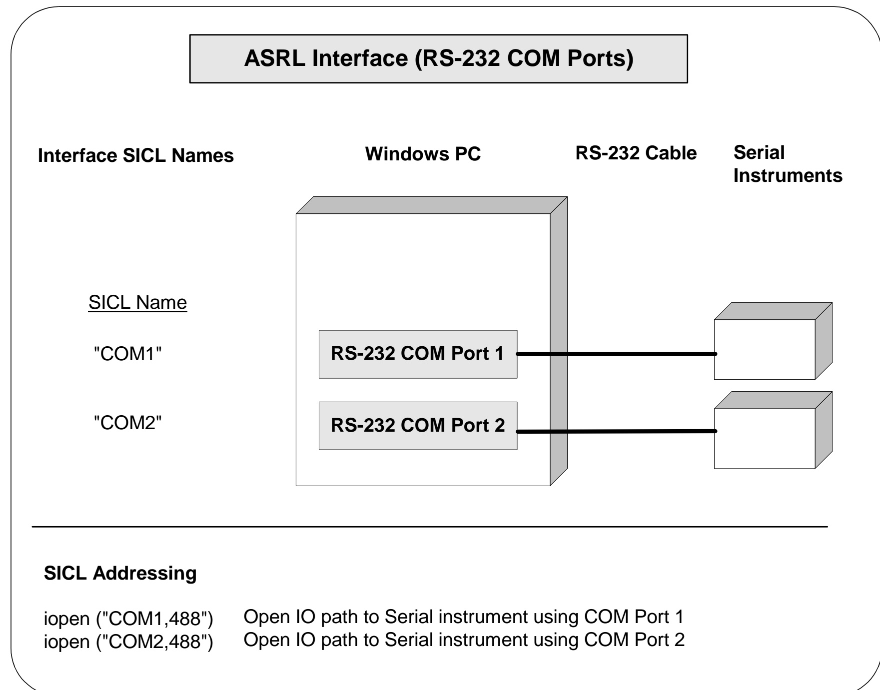
An **IO interface** can be defined as both a hardware interface and a software interface. One function of the Connection Expert utility is to associate a unique interface name with a hardware interface.

SICL uses an **interface ID** or **logical unit number** to identify an interface. This information is passed in the parameter string of the **iopen** function call in a SICL program. Connection Expert assigns a default SICL interface ID and logical unit, as well as other necessary configuration values, when the interface hardware is configured; you can change these values by running the Connection Expert utility. See the *IO Libraries Suite Online Help* for details.

Sample: Configuring RS-232 Interface

The ASRL (RS-232) interface system in the following figure consists of a Windows PC with two RS-232 COM ports, each of which is connected to a single serial instrument via RS-232 cables. Connection

Expert has been used to assign COM Port 1 a SICL name of **COM1** and to assign COM Port 2 a SICL name of **COM2**. Since unique names have been assigned by Connection Expert, you can now use the SICL **iopen** command to open the I/O paths to the GPIB instruments as shown in the figure.



RS-232 Communications Sessions

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 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 Sessions

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.

RS-232 SICL Functions

Table 37 The iserialctrl Functions

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

Table 38 iserialctrl Sets the State for these RS-232 Characteristics

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

Table 38 iserialctrl Sets the State for these RS-232 Characteristics

Request	Characteristic	Settings
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)

Table 39 iserialstat

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

Table 40 iserialstat Captures Status for these RS-232 Characteristics

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_*

Table 40 iserialstat Captures Status for these RS-232 Characteristics

Request	Characteristic	Value
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*

Table 41 Other RS-232 Functions

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.

Table 41 Other RS-232 Functions

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 an RS-232 Device

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 Functions for RS-232 Device Sessions” for details.

The interface logical unit and symbolic name are defined by running the Connection Expert utility. To open Connection Expert, click the Agilent IO Control (**IO** icon on the taskbar) and click **Agilent Connection Expert**. See the *IO Libraries Suite Online Help* for details on this utility.

Some example addresses for RS-232 device sessions:

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

Examples of opening a device session with an RS-232 device:

- C sample:

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

- Visual Basic sample:

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

SICL Functions 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.

Table 42 SICL Functions for RS-232 Device Sessions

Function	Description
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 Functions 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)

Table 42 SICL Functions for RS-232 Device Sessions

ionsrq	Installs a service request handler for this session. Service requests are supported for both device sessions and interface sessions. See “SICL Functions for RS-232 Interface Sessions” in this chapter for details.

Device Session Sample Programs

This section contains two sample programs for RS-232 interface device session programming.

Sample: RS-232 Device Session (C)

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


  /* 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;
  }
```

Sample: RS-232 Device Session (Visual Basic)

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

```
Option Explicit
''''''''''''''''''''''''''''''''''''''''''''''''
'  ser_dev.bas
'  This example program takes a measurement from
` a DVM using a SICL RS-232 device session.
''''''''''''''''''''''''''''''''''''''''''''''''

Sub Main()

  Dim dvm As Integer
  Dim res As Double
  Dim argcount As Integer

  ' Open the multimeter session
  ' "COM1" is the SICL Interface name as defined
  ` in Connection Expert
  ' Change this to the SICL name you have defined
  dvm = iopen("COM1,488")

  ' Set timeout to 10 sec
  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),
    vbExclamation

  ' Close the multimeter session
  Call iclose(dvm)

  ' Tell SICL to cleanup 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.

Addressing RS-232 Interfaces

To create an **interface session** on RS-232, specify the interface logical unit or SICL interface ID in the *addr* parameter of the **iopen** function. The interface logical unit and SICL interface ID are defined by running the Connection Expert utility. To open Connection Expert, click the Agilent IO Control (**IO** icon on the taskbar) and click **Agilent Connection Expert**. See the *IO Libraries Suite Online Help* for details on this utility. Some example addresses for RS-232 interface sessions follow.

Table 43 Sample RS-232 Addresses

COM1	A SICL interface ID
serial	A SICL interface ID
1	An interface logical unit

These code samples open an interface session with the RS-232 interface.

- C sample:

```
INST intf;
intf = iopen ("COM1");
```

- Visual Basic sample:

```
Dim intf As Integer
intf = iopen ("COM1")
```

SICL Functions for RS-232 Interface Sessions

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

Table 44 Implementing Some SICL Functions for RS-232

Functions	Description
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 Connection Expert 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.

Table 44 Implementing Some SICL Functions for RS-232

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**.

Table 44 Implementing Some SICL Functions for RS-232

iserialmclctrl	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.
iserialmclstat	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.

Interface Sessions Sample Programs

This section contains two sample programs for RS-232 interface device session programming.

Sample: RS-232 Interface Session (C)

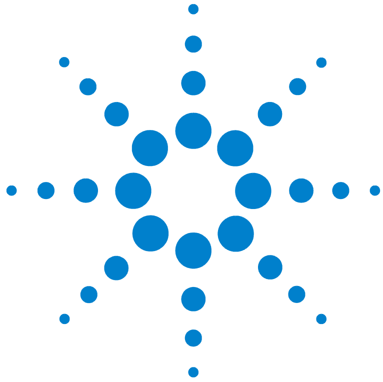
```
/*ser_intf.c
This program gets the current configuration of
the serial port, sets it to 9600 baud, no
parity, 8 data bits, and 1 stop bit, and prints
the old configuration.*/

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

main()
  {
  INST intf; /* interface session id */
  unsigned long baudrate, parity, databits,
    stopbits;
  char *parity_str;


  /* 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 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;
}
```

Sample: RS-232 Interface Session (Visual Basic)

```
Option Explicit
''''''''''''''''''''''''''''''''''''''''''''''''
' set_intf.bas
' This program (1) gets the current
` configuration of the ' serial port; (2) sets
` it to 9600 baud, no parity, 8 data bits, and 1
` stop bit;(3) 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
  ' "COM1" is the SICL Interface name as defined
  ` in Connection Expert
  ' Change this to the SICL Name you have
  ` defined in Connection Expert

  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, vbExclamation

  ' close port
  Call iclose(intf)

  ' Tell SICL to cleanup for this task
  Call siclcleanup

End Sub
```



7
Using SICL with LAN

This chapter shows how to open a communications session and communicate with devices over a Local Area Network (LAN). The sample programs in this chapter can be found in the following locations, if Agilent IO Libraries Suite was installed in the default directory:

For C/C++: `C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\C\SICL`

For Visual Basic:
`C:\Program Files\Agilent\IO Libraries Suite\ProgrammingSamples\VB6\SICL`

The chapter includes:

- LAN Interfaces Overview
- Using Remote Sessions
- Using LAN Interface Sessions
- Using Locks, Threads, and Timeouts

NOTE

This chapter describes SICL programming using the VISA TCPIP interface type to communicate directly with a LAN-connected device, as well as using a remote interface (also known as a LAN client) to emulate a GPIB, serial (ASRL), or USB interface on the local machine to communicate with a LAN-connected device.

See the *Agilent IO Libraries Suite Online Help* for information on how to start and stop the Remote IO Server software, and on how to create and configure LAN interfaces and remote GPIB/USB/serial interfaces.

See the *Connectivity Guide* for detailed information on connecting instruments to a LAN, and for a discussion of network protocols.

Introduction to LAN Interfaces

This section provides an introduction to using SICL with Local Area Network (LAN) interfaces, including:

- LAN and Remote Interfaces Overview
- Considerations when Using SICL with LAN

LAN and Remote Interfaces Overview

This section provides an overview of LAN (Local Area Network) interfaces. A LAN is a way to extend the control of instrumentation beyond the limits of typical instrument interfaces. To communicate with instruments over the LAN, you must first configure a LAN interface or a remote GPIB, USB, or serial interface, using the Agilent Connection Expert.

Direct LAN Connection versus Remote IO Server/Client Connection

Some instruments support direct connection to the LAN. These instruments include an RJ-45 or other standard LAN connector and software support for operating as an independent device on the network. Some of these instruments are Web-enabled, meaning that they host a Web page which you can access over the LAN.

With the Agilent IO Libraries Suite, you can connect to instruments across the LAN even if they do not have direct LAN capability, if they are connected to gateways (such as the Agilent E5810A) or to another PC running the Remote IO Server software.

Refer to the *IO Libraries Suite* and the *Connectivity Guide* for information on connecting and configuring different types of LAN instrument connections.

Remote IO Server/Client Architecture

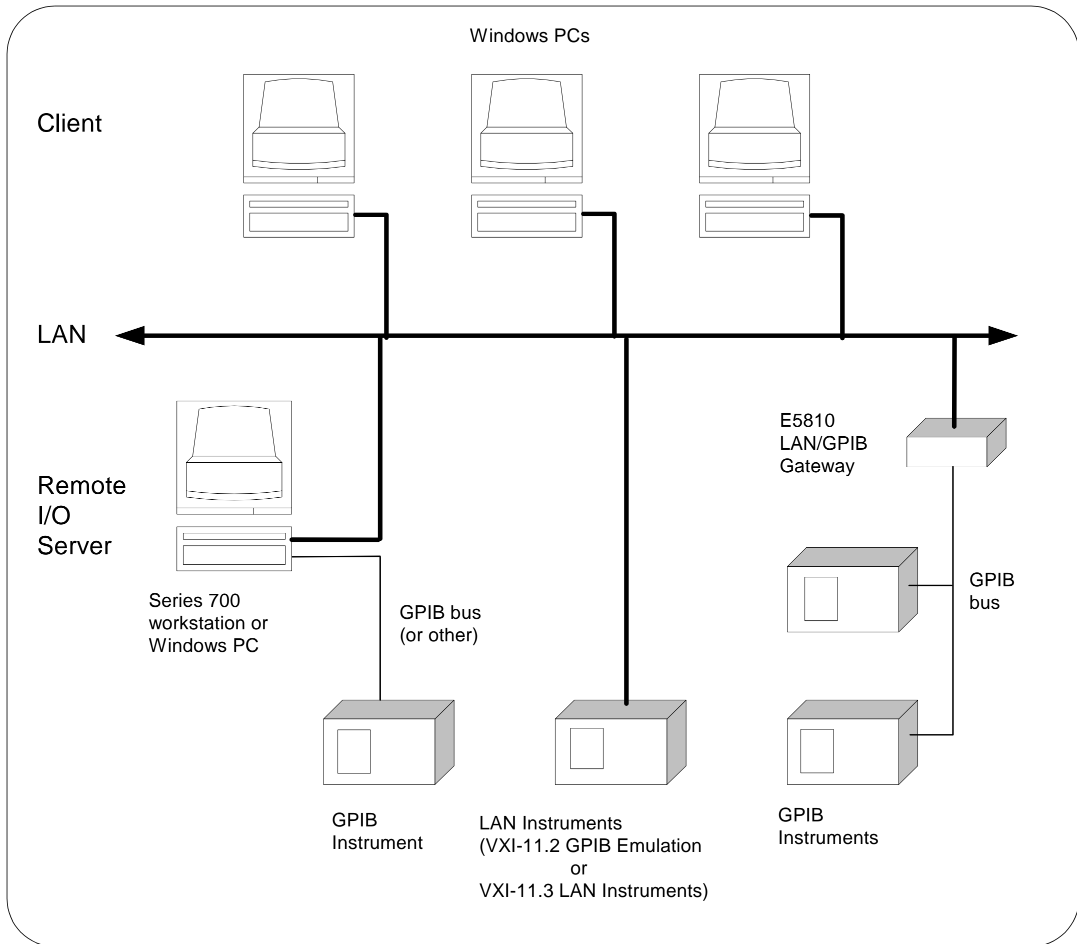
The Remote IO Server and Client software provided with Agilent IO Libraries Suite allows instrumentation to be controlled over a LAN. Using standard LAN connections, instruments can be controlled from computers that do not have special interfaces for instrument control.

Client/server model. The IO Libraries Suite software uses the **client/server** model of computing. Client/server computing refers to a model in which an application (the **client**) does not perform all necessary tasks of the application itself. Instead, the client makes requests of another computing device (the **remote I/O server**) for certain services.

As shown in the following figure, a remote I/O client (a Windows PC) makes VISA requests over the network to a remote I/O server (such as a Windows PC, an E5810 LAN/GPIB Gateway, or a Series 700 HP-UX workstation).

Gateway operation. The remote I/O server is connected to the instrumentation or devices to be controlled. Once the remote I/O server has completed the requested operation on the instrument or device, the remote I/O server sends a reply to the client. This reply contains the requested data and status information that indicates whether or not the operation was successful. The remote I/O server acts as a **gateway** between the LAN software that the client system supports and the instrument-specific interface that the device supports.

.



Considerations when Using SICL with LAN

Specifying Protocol and Socket Number in iopen Calls

As described in the IO Libraries Suite Online Help, you can choose either of two protocols – VXI-11 or SICL-LAN – to associate with a LAN interface. (If you are using a remote GPIB, remote USB, or remote serial interface, you will use Connection Expert to specify a LAN

interface associated with the remote interface. The protocol is defined in the associated LAN interface.) In SICL, you can override this configuration setting by specifying the protocol in the **iopen** string. Some examples are:

- **iopen("lan[machineName]:gpib0,1")** will use the configured default protocol. If AUTO is configured, SICL-LAN protocol will be attempted. If that is not supported, VXI-11 protocol will be used.
- **iopen("lan;auto[machineName]:gpib0,1")** will automatically select the protocol (SICL-LAN if available and VXI-11 otherwise).
- **iopen("lan;sicl-lan[machineName]:gpib0,1")** will use SICL-LAN protocol.
- **iopen("lan;vxi-11[machineName]:gpib0,1")** will use VXI-11 protocol.

The IO Libraries Suite also supports TCP/IP socket reads and writes. To open a socket session, use **iopen("***lan,socketNbr[machineName]"*). For example, **iopen**("*lan,7777[machineName]*") will open a socket connection for socket number 7777 on 'machineName.'

LAN Clients and Threads

You can use multi-threaded designs (with SICL calls 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. See Chapter 3, "Programming with SICL", for more information on using threads in SICL applications. Also see "Using Locks and Threads Over LAN" on page 176 for information on using locks in multi-threaded applications.

SICL LAN Performance

As with other client/server applications on a LAN, when you deploy an application that uses SICL over LAN, you must consider the performance and configuration of the network to which the client and

server will be attached. If the network to be used is neither a dedicated LAN nor 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, that application and/or other network users may experience performance problems due to insufficient bandwidth. This is not unique to SICL over LAN, but is a general design consideration for 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. If you are connecting to a VXI-11 device, you can configure a VXI-11 interface (rather than AUTO) in the Connection Expert utility and connect through it to achieve slightly better **iopen** performance.

SICL LAN Functions

This table summarizes the SICL functions for the LAN interface.

Table 45 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 Remote Sessions

This section provides guidelines to using remote SICL sessions, including:

- Addressing Guidelines
- SICL Function Support
- Sample Programs

Addressing Guidelines

Communicating with a device over a LAN via a TCP/IP or remote GPIB, USB, or serial interface preserves the functionality of the gatewayed interface, with a few exceptions. Thus, most operations over a local interface (such as GPIB connected directly to your controller) can also be performed over a remote interface.

The only portions of your application that must be changed are the addresses passed to the **iopen** calls (unless you use aliases or store those addresses 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.

Creating a Remote Session

To create a **remote session** (also called a **LAN-gatewayed session**), specify the LAN's interface logical unit or interface ID, 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 ID are defined in the Connection Expert utility.

To open Connection Expert, click the Agilent IO Control icon on the taskbar and then click **Agilent Connection Expert**. See the *IO Libraries Suite Online Help* for information on Connection Expert.

Example: Remote Addressing

Some examples of remote SICL addresses follow. If you are using the IP address rather than the hostname of the server machine, you must use the bracket (not the comma) notation.

```
lan,128.10.0.3:gpib  (Incorrect)
lan[128.10.0.3]:gpib  (Correct)
```

Table 46 Examples of LAN Addressing

Address	Description
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**. The default LAN protocol set when the LAN interface was configured with Connection Expert will be used.
lan;vxi-11[instserv]:GPIB,7	A device address corresponding to the device at primary address 7 on the GPIB interface attached to the machine named **instserv**. The VXI-11 protocol (TCP/IP Instrument protocol) will be used.
lan;sicl-lan [instserv]:GPIB,7	A device address corresponding to the device at primary address 7 on the GPIB interface attached to a machine named **instserv**. The SICL-LAN protocol will be used.
lan;auto[instserv]:GPIB,7	A device address corresponding to the device at primary address 7 on the GPIB interface attached to a machine named **instserv**. The SICL-LAN protocol will be used if the server supports it. Otherwise, the VXI-11protocol will be used.
lan;default[instserv]:GPIB,7	A device address corresponding to the device at primary address 7 on the GPIB interface attached to a machine named **instserv**. The default LAN protocol set when the lan interface was configured with Connection Expert will be used. This is the same as not specifying a protocol.

Table 46 Examples of LAN Addressing

lan[instserv.agilent.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 agilent.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[instserv]:GPIB2	An interface address corresponding to the **GPIB2** interface attached to the machine named **instserv**.
30,instserv: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 **instserv**. (30 is the default logical unit for LAN.)
lan[instserv]:GPIB,cmdr	A commander session with the **GPIB** interface attached to the machine named **instserv** (assuming the server supports GPIB commander sessions).
lan[instserv]:COM1	An interface address corresponding to the RS-232 **COM1** interface attached to the machine named **instserv**.
lan[instserv]:COM1,488	A device address corresponding to an RS-232 device attached to the machine named **instserv**.
lan[instserv]:usb0[2391::1031 ::SN_041001::0]	A device address corresponding to a USB device attached to the machine named **instserv**.
lan[instserv]:UsbDevice1	A device address corresponding to a USB device attached to the machine named **instserv**. The alias name **UsbDevice1** is defined on the machine named **instserv**.

SICL Function Support

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**.

Table 47 Relationships Between SICL Functions

Address	Session Type	Interface Type (VXI-11 Protocol)	Interface Type (SICL-LAN Protocol)	Gateway Type
lan	I_SESS_INTF	I_INTF_LAN	I_INTF_NONE	I_INTF_NONE
lan[instserv]:inst0	I_SESS_DEV	I_INTF_LAN	I_INTF_USRDEF	I_INTF_LAN
lan[instserv]:gpib0	I_SESS_INTF	I_INTF_GPIB	I_INTF_GPIB	I_INTF_LAN
lan[instserv]:gpib0,7	I_SESS_DEV	I_INTF_GPIB	I_INTF_GPIB	I_INTF_LAN
gpib0	I_SESS_INTF	I_INTF_GPIB	I_INTF_GPIB	I_INTF_NONE
gpib0,7	I_SESS_DEV	I_INTF_GPIB	I_INTF_GPIB	I_INTF_NONE

Remote Interface Support

A gatewayed session to a remote interface provides the same SICL function support as if the interface were local, with the following exceptions or qualifications.

Table 48 Exceptions to Remote Interface Support

Type of Functions	SICL Functions NOT Supported
SICL functions *not* supported over LAN using either protocol	iblockcopy, imap, imapinfo, ipeek, ipoke, ipopfifo, ipushfifo, iunmap, iblockmovex, imapx, iunmapx, ipeekx, ipokex, iunmapx
SICL functions, *in addition to those listed above*, *not* supported with the VXI-11 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 VXI-11 (TCP/IP instrument protocol), the remote address strings *must* follow the VXI-11 naming conventions – `gpib0,` `gpib1,` etc. For example:

```
gpib0,7
gpib1,7,2
gpib2
vxi0, vxi1, etc. (for example: vxi0,8 or vxi0)
```

However, since the interface IDs at the remote server may be configurable, this conformance is not guaranteed. Correct behavior of **iremote** and **iclear** depend on the correct address strings being used. When **iremote** is executed over the VXI-11 protocol, **iremote** sends the LLO (local lockout) message in addition to placing the device in the remote state.

LAN Timeout Functions

Any of the following functions may time out over LAN, even those functions that cannot time out 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.

Table 49 How SICL Functions Perform for LAN Gatewayed Devices

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.

Sample Programs

Two sample programs for LAN-gatewayed sessions follow, one for C and one for Visual Basic 6.0.

Sample: LAN-gatewayed Session (C) This sample program opens a GPIB device session via a LAN-to-GPIB gateway. This sample is the same as the sample in *Chapter 4 - Using SICL with GPIB*, except the addresses passed to the **iopen** calls are modified. The addresses in this sample 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]:gpib0,9,3");
  sw = iopen ("lan[instserv]:gpib0,9,14");
  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);
}
```

Sample: LAN-gatewayed Session (Visual Basic 6.0)

This sample program opens a GPIB device session via a LAN-to-GPIB gateway.

```
Option Explicit
''''''''''''''''''''''''''''''''''''''''''''''''
' landev.bas
' This example program opens a GPIB device
` session via a LAN-to-GPIB gateway. The
` addresses in this example assume a machine
` with hostname 'instserv' is acting as a
` LAN-to-GPIB gateway.
''''''''''''''''''''''''''''''''''''''''''''''''

Sub Main()

  Dim dvm As Integer, sw As Integer
  Dim nargs As Integer, I As Integer
  Dim actual As Long
  Dim res As String * 20

  ' 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]:gpib0,9,3")
  sw = iopen("lan[intserv]:gpib0,9,14")
  Call itimeout(dvm, 10000)
  Call itimeout(sw, 10000)

  ' set up the trigger
  nargs = iwrite(sw, "TRIG:SOUR BUS" + Chr$(10)
    + Chr$(0), 14, 1, actual)

  ' set up scan list
  nargs = iwrite(sw, "SCAN (@100:103)" +
    Chr$(10) + Chr$(0), 15, 1, actual)
  nargs = iwrite(sw, "INIT" + Chr$(10) +
    Chr$(0), 5, 1, actual)

  For I = 1 To 4 Step 1
    ' Take a measurement
    nargs = iwrite(dvm, "MEAS:VOLT:DC?" +
      Chr$(10)+ Chr$(0), 14, 1, actual)

    ' Read the results
    nargs = iread(dvm, res, 20, 0&, actual)

    ' Print the results
    MsgBox "Channel " & I & " result:  " + res &
      vbCrLf

    ' Trigger switch
    nargs = iwrite(sw, "TRIG" + Chr$(10) +
      Chr$(0), 5, 1, actual)
  Next I

  Call iclose(dvm)
  Call iclose(sw)

  Exit Sub

ErrorHandler:

  ' Display the error message in the txtResponse
  ' TextBox.

  MsgBox "*** Error : " + Error$
```

```
  ' Close the device session if iopen was
  ` successful.

  If dvm <> 0 Then
    Call iclose(dvm)
  End If

  If sw <> 0 Then
    Call iclose(sw)
  End If

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” on page 177.)

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 SICL interface ID are defined by the Connection Expert utility.

To open Connection Expert, click the Agilent IO Control **IO** icon on the taskbar and then click **Agilent Connection Expert**. See the *IO Libraries Suite Online Help* for information on Connection Expert. Some examples of SICL interface IDs for LAN interfaces follow.

Table 50 SICL Interface ID Examples

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

These SICL functions are *not* supported over LAN interface sessions; they 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.

Table 51 SICL Functions for 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 or LAN interfaces.

Using Locks, Threads, and Timeouts

This section gives guidelines to use locks, threads, and timeouts over LAN, including:

- Using Locks and Threads Over LAN
- Using Timeouts Over LAN

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.

Scenarios to Avoid

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*.

- *Avoid* 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**.
- *Avoid* using a device symbolic name, or alias, 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.
- *Avoid* using a remote interface’s logical unit (such as “7”) in one call while using the remote interface’s SICL interface ID (such as “gpib”) in another.
- *Avoid* using **igetintfsess** to open an interface session (which internally uses the logical unit to identify the remote interface) while opening the interface with its SICL interface ID for another session.

Recommended Usage

You can avoid each of the above scenarios 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 instead of the SICL interface ID to specify the interface.

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**. You can also use Win32 synchronization techniques 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 remote I/O 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.

Client/Server Operation

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 Connection Expert.

Once the application calls **ilantimeout**, the automatic LAN timeout adjustment is turned off.

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.*

Default LAN Timeout Values

Connection Expert 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**.

Table 52 LAN Software Timeout Values

LAN maximum 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).

Table 52 LAN Software Timeout Values

Client delta timeout	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.

Timeout Algorithm

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 Delta Timeout is greater than the current LAN timeout. In that case the LAN timeout will be set to the SICL timeout plus the Client Delta Timeout.
- 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 Delta Timeout specified via Connection Expert for portions of the **iopen** operation. The timeout value for TCP connection establishment is not affected by the Client Delta Timeout.

To change the timeout values:

1. Exit any applications that use SICL.
2. Run the Connection Expert utility. (Click the Agilent IO Control and then click **Agilent Connection Expert**.)

3 Click on the TCPIP interface shown in the explorer view, then click **Change Properties...** in the properties pane on the right.

4 Change the LAN Maximum Timeout and/or Client Delta Timeout value(s) and click **OK** to save the changes.

5 Restart your application(s).

Timeouts in Multi-threaded Applications

If you want to manually set the client-side timeout in an application using multiple threads, be aware that **ilantimeout** may itself time out due to contention for the LAN subsystem, in cases 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 you call **ilantimeout** only when no other LAN I/O is occurring, such as immediately after session creation (**iopen**).

If you use the no-wait value 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 two minutes. If another thread is using the LAN interface for longer than two minutes, the no-wait operation will time out.

Timeout Configurations to Be Avoided

The LAN timeout used by the client should always be 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 network problems or clients that have ceased operation abruptly, and subsequently release resources associated with those clients, such as locks.

Using the smallest possible timeout for your application will maximize the server's responsiveness to dropped connections, including dropped connections that result from the client application being terminated abnormally. You can set a value less than infinity by setting the LAN Maximum Timeout configuration value in the Connection Expert utility.

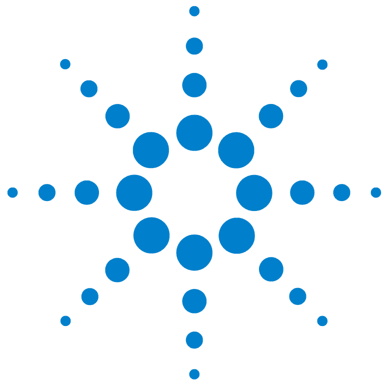
Even if your application uses the SICL default of infinity, or if **itimeout** is used to set the timeout to infinity, by setting the LAN Maximum Timeout value to some reasonable number of seconds, you allow the server to time out, detect network trouble, and release resources.

Application Terminations and Timeouts

If an application is stopped in the middle of a SICL operation performed at the remote I/O server, the server continues to try the operation until the server's timeout is reached. By default, the remote I/O server associated with an application that is using a timeout of infinity and is stopped may not discover that the client is no longer running for two minutes. If you are using a server other than the Agilent Remote IO Server on Windows or the Agilent E5810 LAN/GPIB gateway, check your server's documentation for its default behavior.

If your application uses **itimeout** 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" (unresponsive). If this situation occurs, configure the LAN interface (via the Client Delta Timeout value set with Connection Expert) or the Remote IO Server (via its Server Timeout value) to use a shorter timeout value.

If you must use long timeouts, you may reset the server to regain server response. You can reset a remote I/O server by logging into the server system and stopping the Remote IO Server software that is running. This will affect all clients connected to the server. See *"Appendix B: Troubleshooting SICL Programs"* for more details. Also, see the documentation on the server you are using for methods to reset the server.



8
Using SICL with USB

This chapter provides guidelines for SICL programming of USB instruments that conform to USBTMC (Universal Serial Bus Test and Measurement Class) and/or USBTMC-USB488 (Universal Serial Bus Test and Measurement Class, Subclass USB488 Specification).

The chapter contents are:

- USB Interfaces Overview
- Communicating with a USB Instrument Using SICL

USB Interfaces Overview

USBTMC/USBTMC-USB488 instruments are detected and automatically configured by Agilent IO Libraries Suite when they are plugged into the computer. The *IO Libraries Suite Online Help* describes the USB instrument configuration process in more detail.

NOTE

Do not confuse the Agilent 82357 USB/GPIB Interface with a USBTMC device. The 82357 is automatically configured as a GPIB interface, not as a USBTMC device, when it is plugged into the computer. Only USBTMC/USBTMC-USB488 devices will be configured as USB devices by Agilent IO Libraries Suite.

Communicating with a USB Instrument Using SICL

Each USBTMC device can be uniquely identified by a set of four parameters. These parameters are described in the following table.

Table 53 USBTMC Device Parameters

Parameter	Data Type	Example Value	Default Value
Manufacturer ID	16-bit unsigned integer	2391	n/a
Model Code	16-bit unsigned integer	1031	n/a
Serial Number	String (128 characters max)	SN_001001	n/a
USBTMC Interface Number	8-bit unsigned integer	0	0

When a USBTMC instrument is attached to the computer, Agilent IO Libraries Suite automatically configures a USB interface with the name **usb0** if one does not already exist. (See the *IO Libraries Suite Online Help* for more details.) A dialog box is also displayed, showing an **alias** (which you can change) and the four unique USB parameters for the device.

To establish communications with a USB device using SICL, you can use either the full SICL resource string for the device or use the alias. Using the alias is recommended, for reasons described below.

Using the full SICL resource string to open a USB instrument, the **iopen** call would look like this:

```
id = iopen("usb0[2391::1031::SN_001001::0]");
```

Since in this example the USBTMC interface ID has the default value of 0, it does not have to be specified. The **iopen** call would then look like this:

```
id = iopen("usb0[2391::1031::SN_001001]");
```

Following is a summary of the components of this call.

Table 54 Summary of Full-String iopen Call

Value	Description
usb0	the SICL name for the USB interface
2391	Manufacturer ID
1031	Model Code
SN_001001	Serial Number
0	USBTMC Interface Number

This address string uniquely identifies the USB device. The values needed for the resource string are displayed in a dialog box when the device is plugged into the computer. The same values can also be obtained by running the Connection Expert utility and selecting the USB interface in the explorer view; the values will be shown in the properties view (right pane of the Connection Expert window).

To simplify the way a USB device is identified, SICL also provides an **alias** which can be used in place of this resource string. The first USB device that is plugged in is assigned a default alias of **UsbDevice1**. Additional devices are assigned aliases of **UsbDevice2**, **UsbDevice3**, etc. You can modify the alias to one of your choosing at the time a device is plugged in, or by running the Connection Expert utility and modifying the properties of the USB interface.

Note that the Connection Expert displays (and allows editing of) VISA aliases, not SICL aliases. However, Connection Expert creates a SICL alias to correspond to each VISA alias, so if you do not use other tools to edit your aliases, the VISA and SICL aliases in your test system will be identical.

Although the case of an alias is preserved, case is ignored when the alias is used in place of the full resource string in an **iopen** call. For example, `UsbDevice1`, `usbdevice1` and `USBDEVICE1` all refer to the same device.

Using the alias, an **iopen** call would look like this:

```
id = iopen("UsbDevice1");
```

As you can see, this is much simpler than having to use the full resource string for a USB device.

Using the alias name in a program also makes it more portable. For example, two identical USB function generators have different resource strings because they have different serial numbers. If these function generators are used in two different test systems and you use the full resource string to access the function generator in the test program, you cannot use that same program for both test systems, since the function generators' full resource strings are different. By using the alias name in the program, however, you can use the same program in both test systems. All you need to do is make sure the same alias name is used for the function generator in both systems.

Operations Supported on All USBTMC Devices

The following USB-specific SICL functions can be used on all USBTMC and USBTMC-USB488 device sessions. (See the *SICL Online Help* for specific information on these functions.)

Table 55 Operations Supported on All USBTMC Devices

Function Name	Action
iusbctrl	Used to set parameters affecting the USB device.
iusbgetcapabilitie s	Returns a structure containing capabilities information about the USB device.
iusbgetinfo	Returns a structure containing general information about the USB device.
iusbstat	Used to retrieve the settings of parameters affecting the USB device.

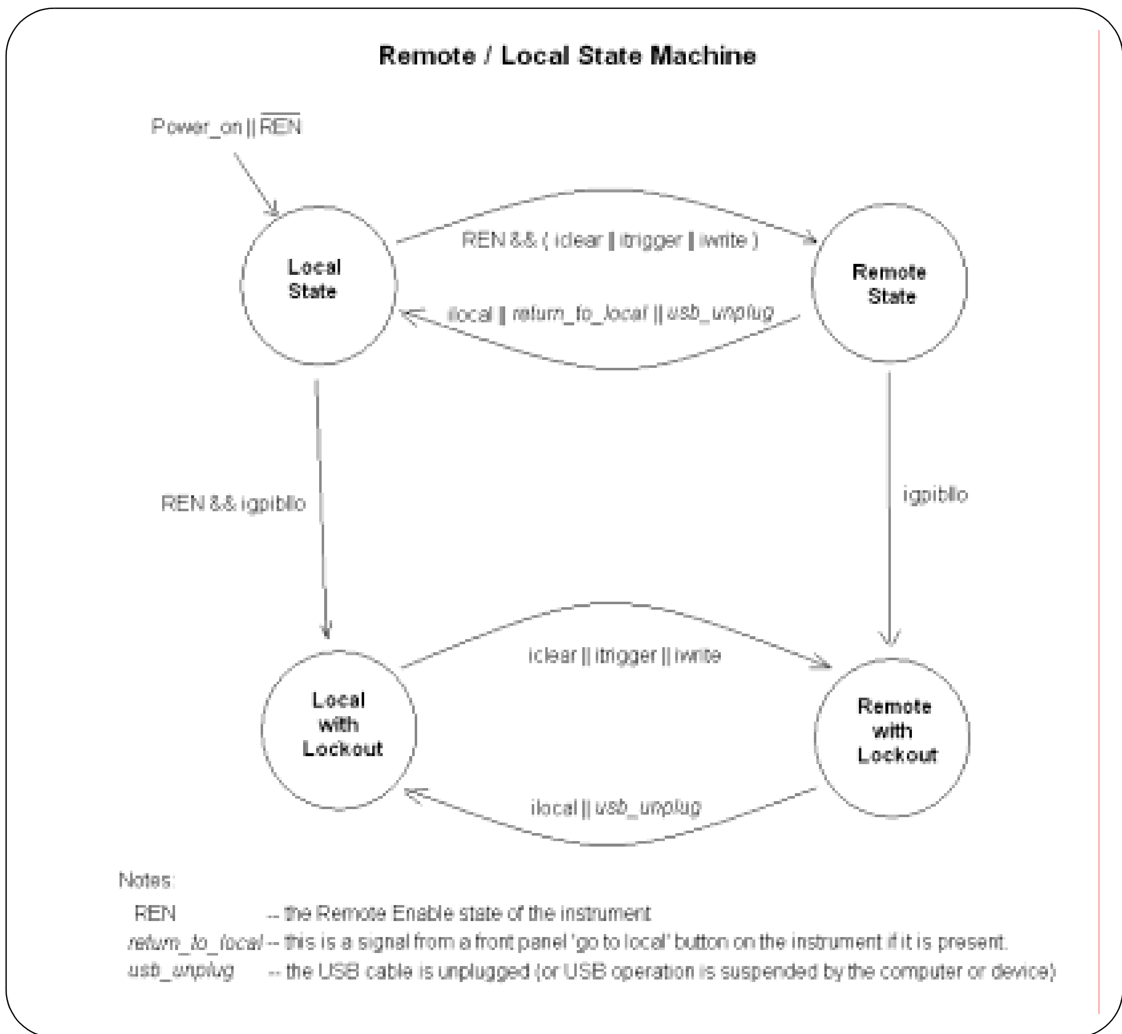
Interrupts are not supported on USBTMC or on USBTMC-USB488 devices.

Operations Supported Only on USBTMC-USB488 Devices

The *iusbgetcapabilities* function can be used to determine if a device supports the USBTMC-USB488 protocol. See the *SICL Online Help* for specific information on this function and the definition of the structure that it returns. If the *bcdUSB488* structure element is non-zero, the device implements the USBTMC-USB488 protocol. The *intf488Capabilities* and *dev488Capabilities* bit masks in this structure provide the details of what the device supports (e.g. REN Control, Triggering, SCPI commands, etc.)

SRQs (*ionsrq*) and triggers (*itrigger*) are supported only on USBTMC-USB488 devices. They are not supported on USBTMC devices that do not implement the USBTMC-USB488 protocol.

On USBTMC-USB488 devices that support REN control, the following state diagram shows how state transitions are made using various SICL functions.



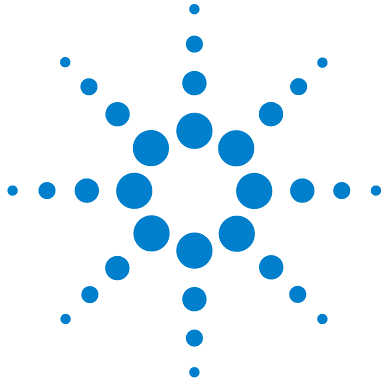
The following SICL functions are used to control the Remote/Local state transitions in USBTMC-USB488 devices sessions. (See the *SICL Online Help* for specific information on these functions.)

Table 56 SICL Functions for Remote/Local State Transition

Function Name	Action
igpibllo	Locks out the front panel interface of the device (if REN is true).
igpibibrenctl iremote	Sets the REN (remote enable) state: igpibrenctl(id, 0) sets REN false. igpibrenctl(id, 1) sets REN true. iremote(id) sets REN true.
ilocal	Enables the front panel interface of the device.
iremote	Sets the device REN (remote enable) state to true.

NOTE

Although *igpibllo* and *igpibrenctl* are documented as GPIB-specific functions that are only valid on interface sessions, these functions can be called on USBTMC-USB488 device sessions.



Appendix A: SICL Library Information

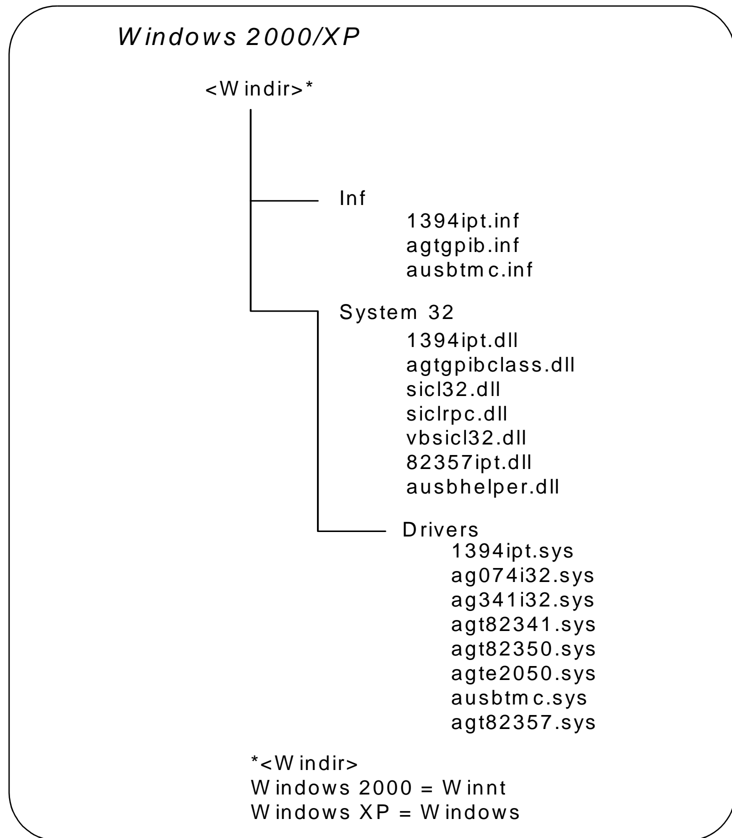
SICL Library Information

This appendix provides information on SICL software files and Windows system interactions.

File System Information

This section describes SICL file system information for Windows systems.

All SICL files are installed in the base directory specified by the person who installs Agilent IO Libraries Suite, with the exception of several common files that Windows must be able to locate. On Windows 2000, the following files are copied to the `Windows` subdirectory. On Windows XP, the files are copied to `Winnt`.



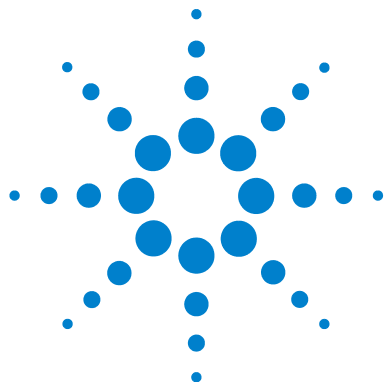
The Registry

Agilent IO Libraries Suite places the following keys in the Windows registry under HKEY_LOCAL_MACHINE:

```
Software\Agilent\IO Libraries\CurrentVersion
Software\Agilent\IO Libraries Suite\CurrentVersion
```

Also, if the Remote IO Server is configured, the following key will be created under HKEY_LOCAL_MACHINE if it did not previously exist:

```
Software\Microsoft\Windows\CurrentVersion\
RunServices
```



Appendix B: Troubleshooting SICL Programs

Troubleshooting SICL Programs

This appendix contains a description of SICL error codes and provides guidelines for troubleshooting common problems with SICL. The chapter contents are:

- SICL Error Codes
- Common Windows Problems
- Common RS-232 Problems
- Common LAN Problems

See the *Agilent IO Libraries Suite Online Help* and the *Agilent USB/LAN/GPIB Interfaces Connectivity Guide* for additional troubleshooting guidelines.

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 is logged.

SICL logs internal messages as events that you can view by clicking the Agilent IO Control (on the taskbar) and then clicking **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.

Table 57 List of SICL Error Codes

Error Number	Error Code	Error String	Description
23	I_ERR_ABORTED	Externally aborted	A SICL call was aborted by external means.
3	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 Connection Expert.
24	I_ERR_BADCONFIG	Invalid configuration	An invalid configuration was identified when calling `iopen`.
13	I_ERR_BADFMT	Invalid format	Invalid format string specified for **iprintf** or **iscanf**.
4	I_ERR_BADID	Invalid INST	The specified **INST** *id* does not have a corresponding **iopen**.
19	I_ERR_BADMAP	Invalid map request	The **imap** call has an invalid map request.
28	I_ERR_BUSY	Interface is in use by non-SICL process	The specified interface is busy.
14	I_ERR_DATA	Data integrity violation	The use of CRC, Checksum, and so forth imply invalid data.
128	I_ERR_INTERNAL	Internal error occurred	SICL internal error.
129	I_ERR_INTERRUPT	Process interrupt occurred	A process interrupt (signal) has occurred in your application.
21	I_ERR_INVLADDR	Invalid address	The address specified in **iopen** is not a valid address (for example, “hpib,57”).
17	I_ERR_IO	Generic I/O error	An I/O error has occurred for this communication session.
11	I_ERR_LOCKED	Locked by another user	Resource is locked by another session (see **isetlockwait**).
27	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.
25	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.

Table 57 List of SICL Error Codes (continued)

Error Number	Error Code	Error String	Description
6	I_ERR_NOCONN	No connection	Communication session has never been established, or connection to remote has been dropped.
20	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.
0	I_ERR_NOERROR	No Error	No SICL error returned; function return value is zero (0).
10	I_ERR_NOINTF	Interface is not active	Tried to specify an interface session when it is not active, available, or does not exist.
12	I_ERR_NOLOCK	Interface not locked	An **iunlock** was specified when device was not locked.
7	I_ERR_NOPERM	Permission denied	Access rights violated.
9	I_ERR_NORSRC	Out of resources	No more system resources available.
22	I_ERR_NOTIMPL	Operation not implemented	Call not supported on this implementation. The request is valid, but not supported on this implementation.
8	I_ERR_NOTSUPP	Operation not supported	Operation not supported on this implementation.
18	I_ERR_OS	Generic O.S. error	SICL encountered an operating system error.
16	I_ERR_OVERFLOW	Arithmetic overflow	Arithmetic overflow. The space allocated for data may be smaller than the data read.
5	I_ERR_PARAM	Invalid parameter	The constant or parameter passed is not valid for this call.
2	I_ERR_SYMNAME	Invalid symbolic name	Symbolic name passed to `iopen` not recognized.

Table 57 List of SICL Error Codes (continued)

Error Number	Error Code	Error String	Description
1	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.
15	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**.
26	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

Table 58 Windows Errors

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 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++ 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.

Table 59 Common RS-232 Problems

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 A - SICL Library Information* for RS-232 cabling information. 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.

Table 59 Common RS-232 Problems

Data Lost During Large Transfers	Check: Flow control setting match Full/half duplex for 3-wire connections Cabling is correct for hardware handshaking

Common LAN Problems

NOTE Both the LAN client and LAN server may log messages to the **Event Viewer** 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.agilent.com

Pinging instserv.agilent.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.agilent.com[128.10.0.3] with 32
bytes of data:
Request timed out.
Request timed out.
Request timed out.
Request timed out.
```

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 Remote IO Server software 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 Remote IO 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 Delta Timeout configuration value via the Connection Expert 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 Lan Maximum Timeout configuration value via the Connection Expert 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 Delta Timeout configuration value is large enough (via the Connection Expert 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 *IO Libraries Suite Online Help* for details on how 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 *IO Libraries Suite Online Help* for information on using the Connection Expert utility 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 *IO Libraries Suite Online Help*.

This action will affect all connected clients, even those that may still be operational. If informational logging has been enabled using the Connection Expert utility, connected clients can be determined by log entries in the **Event Viewer** utility.

rpcinfo Fails - cannot 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 *IO Libraries Suite Online Help* for details on how to start and stop the LAN server.

rpcinfo Fails - program 395180 is not available

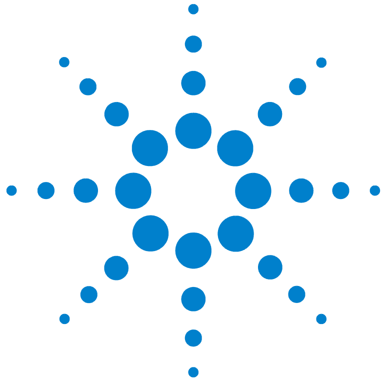
An **rpcinfo -t** *server_hostname* **395180 1** returns the following message:

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

After you attempt to stop a LAN server via either **Ctrl+C** or the Windows Close button (the **x** 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.



Glossary

access board

The GPIB interface to which a particular device is connected.

Active Controller

See "Controller in Charge".

address

A string (or other language construct) that uniquely locates and identifies a resource. VISA defines an ASCII-based grammar that associates address strings with particular physical devices or interfaces and VISA resources.

Agilent 488

An I/O library provided in Agilent IO Libraries Suite for compatibility with existing test & measurement programs that were developed using National Instruments' NI-488 or other similar libraries. Agilent 488 supports communication with GPIB devices and interfaces, but does not support USB, LAN, RS-232, or VXI communications.

alias

See **VISA alias**.

API

Application Programming Interface. The interface that a programmer sees when creating an application. For example, the VISA API consists of the sum of all of the operations, attributes, and events of each of the VISA ResourceClasses.

attribute

In VISA and SICL, a value that indicates the operational state of a resource. Some attributes can be changed; others are read-only.

board

A GPIB interface. It may be a physical board, an adapter (such as the 82357 USB/GPIB adapter), or a remote GPIB interface.

board descriptor

A handle, returned from ibfind, that uniquely identifies a GPIB interface (board) in Agilent 488 original API calls. Also called an interface descriptor or board unit descriptor.

board-level

Refers to Agilent 488 functions that operate on an interface (board), rather than on a device.

bus error

An error that signals failure to access an address. Bus errors occur in conjunction with low-level accesses to memory, and usually involve hardware with bus mapping capabilities. Bus errors may be caused by non-existent memory, a non-existent register, an incorrect device access, etc.

bus error handler

Software that runs when a bus error occurs.

CIC

Controller in Charge.

command bytes

GPIB commands encoded as individual bytes. Also called **GPIB commands** or **interface messages**.

commander

In test-system architectures, a device that has the ability to control another device. In a specialized case, a commander may also be the device that has sole control over another device (as with the VXI Commander/Servant hierarchy).

commander session

A session that communicates to the interface's commander. Commander sessions are used when an interface is in a non-Controller role.

communication channel

A communication path between a software element and a resource. In VISA, "communication channel" is synonymous with "session." Every communication channel in VISA is unique.

Connection Expert

An Agilent software utility that helps you quickly establish connections between your instruments and your PC. It also helps you troubleshoot connectivity problems. Connection Expert is part of the Agilent IO Libraries Suite product.

Controller

A device (typically a computer) used to communicate with another device or devices (typically instruments). The Controller is in charge of communications and device operation; it controls the flow of communication and performs addressing and other bus management functions.

Controller in Charge

The device currently in control of the GPIB.

device

A unit that receives commands from a Controller. A device is typically an instrument, but can also be a computer acting in a non-Controller role or another peripheral such as a printer or plotter. In VISA, a device is logically represented by the association of several VISA resources.

device descriptor

A handle, returned from **ibdev** or **ibfind**, that uniquely identifies a device in Agilent 488 original API calls. Also called a **device unit descriptor**.

device driver

Software code that communicates with a device: for example, a printer driver that communicates with a printer from a PC. A device driver may either communicate directly with a device by reading to and writing from registers, or it may communicate through an interface driver.

device session

A session that communicates as a Controller with a single, specific device such as an instrument.

device-level

Refers to Agilent 488 functions that operate on a device (instrument), rather than on an interface.

direct I/O

Programmatic communication with instruments not involving an instrument driver. Direct I/O may be accomplished by using an IO Library (VISA, VISA COM, SICL, or Agilent 488) or by using direct I/O tools such as those provided by Agilent VEE.

driver

See **instrument driver** and **device driver**.

explorer view

The tree view within the Connection Expert window that shows all devices connected to a test system.

handler

A software routine that responds to an asynchronous event such as an SRQ or an interrupt.

instrument

A device that accepts commands and performs a test and measurement function.

instrument driver

Software that runs on a computer to allow an application to control a particular instrument.

Interactive IO

An Agilent application that allows you to interactively send commands to instruments and read the results. Interactive IO is part of the Agilent IO Libraries Suite product.

interface

A connection and medium of communication between devices and controllers. Interfaces include mechanical, electrical, and protocol connections.

interface descriptor

A handle, returned from **ibfind**, that uniquely identifies a GPIB interface (board) in Agilent 488 original API calls. Also called a **board descriptor** or **board unit descriptor**.

interface driver

Software that communicates with an interface. The interface driver also handles commands used to perform communications on an interface.

interface messages

GPIB commands encoded as individual bytes. Also called **GPIB commands** or **command bytes**.

interface session

A session that communicates and controls parameters affecting an entire interface.

interrupt

An asynchronous event that requires attention and actions that are out of the normal flow of control of a program.

IO Control

The icon in the Windows notification area (usually the lower right corner of your screen). The IO Control gives you access to Agilent I/O utilities such as Connection Expert, Agilent I/O documentation, and VISA options.

IO Libraries

Application programming interfaces (APIs) for direct I/O communication between applications and devices. There are four Agilent IO Libraries in the Agilent IO Libraries Suite: VISA, VISA COM, SICL, and Agilent 488.

Listener

A device that can receive data from the bus when instructed (addressed to listen) by the System Controller.

lock

A state that prohibits other users from accessing a resource such as a device or interface.

logical unit

A number associated with an interface. A logical unit, in SICL and Agilent VEE, uniquely identifies an interface. Each interface on the controller must have a unique logical unit.

mapping

An operation that returns a reference to a specified section of an address space and makes the specified range of addresses accessible to the requester. This function is independent of memory allocation.

non-Controller role

A computer is in a non-Controller role when it acts as a device communicating with a computer that is in a Controller role.

notification area

The area on the Windows taskbar where notifications are posted, typically in the lower right corner of the screen. Also called **taskbar notification area** or **Windows notification area**.

operation

A defined action that can be performed on a resource.

primary VISA

The VISA installation that controls the visa32.dll file. The primary VISA will be used by default in VISA applications. See also **secondary VISA**.

process

An operating system component that shares a system's resources. A single-process computer system allows only a single program to execute at any given time. A multi-process computer system allows multiple programs to execute simultaneously, each in a separate process environment.

programming alias

See **VISA alias**.

refresh

In Connection Expert, the action that invokes the discovery mechanism for detecting interfaces and instruments connected to your computer. The explorer view is then refreshed to show the current, discovered state of your test system.

register

An address location that contains a value that represents the state of hardware, or that can be written into to cause hardware to perform a specified action or to enter a specified state.

resource (or resource instance)

In VISA, an implementation of a resource class (in object-oriented terms, an instance of a resource class). For example, an instrument is represented by a resource instance.

resource class

The definition of a particular resource type (a class in object-oriented terms). For example, the VISA Instrument Control resource classes define how to create a resource to control a particular capability of a device.

resource descriptor

A string, such as a VISA resource descriptor, that specifies the I/O address of a device.

SCPI

Standard Commands for Programmable Instrumentation: a standard set of commands, defined by the SCPI Consortium, to control programmable test and measurement devices in instrumentation systems.

secondary VISA

A VISA installation that does not install visa32.dll in the standard VISA location. The secondary VISA installation names its VISA DLL with a different name (agvisa32.dll) so that it can be accessed programmatically. The primary VISA will be used by default in VISA applications. See also **primary VISA**.

session

VISA term for a communication channel. An instance of a communications path between a software element and a resource. Every communication channel in VISA is unique.

SICL

Standard Instrument Control Library. SICL is an Agilent-defined API for instrument I/O. Agilent SICL is one of the IO Libraries installed with Agilent IO Libraries Suite.

side-by-side

A side-by-side installation allows two vendors' implementations of VISA to be used on the same computer. See also **primary VISA** and **secondary VISA**.

SRQ

An IEEE-488 Service Request. This is an asynchronous request (an interrupt) from a remote device that requires service. In GPIB, an SRQ is implemented by asserting the SRQ line on the GPIB. In VXI, an SRQ is implemented by sending the Request for Service True event (REQT).

Standby Controller

A device or interface that has Controller capability, but is not currently the Active Controller.

status byte

A byte of information returned from a remote device that shows the current state and status of the device. If the device follows IEEE-488 (GPIB) conventions, bit 6 of the status byte indicates whether the device is currently requesting service.

symbolic name

A name corresponding to a single interface. This name uniquely identifies the interface on this Controller or gateway. When there is more than one interface on the Controller or gateway, each interface must have a unique symbolic name.

System Controller

One Controller on a GPIB is the System Controller. This is a master Controller; it has the ability to demand control and to assert the IFC (Interface Clear) and REN (remote enable) lines.

system tray

See **notification area**.

Talker

A device that transmits data onto the bus when instructed (addressed to talk).

task guide

The information and logic represented in the left pane of the Connection Expert window. The task guide provides links to actions and information that help guide you through the most common I/O configuration tasks.

taskbar notification area

See **notification area**.

test system

An entire test setup including a controller (often a PC), instruments, interfaces, software, and any remote controllers, instruments, and interfaces that are configured to be used as part of the system.

thread

An operating system object that consists of a flow of control within a process. A single process may have multiple threads, each having access to the same data space within the process. Each thread has its own stack, and all threads may execute concurrently (either on multiple processors, or by time-sharing a single processor).

ViFind32

A console application that uses the viFindRsrc and viFindNext VISA functions to enumerate all resources visible to VISA. This application is useful for verifying that all expected interfaces have been configured by Connection Expert, and that the expected devices have been attached. ViFind32 is part of the Agilent IO Libraries Suite.

virtual instrument

A name given to the grouping of software modules (such as VISA resources with any associated or required hardware) to give them the functionality of a traditional stand-alone instrument. Within VISA, a virtual instrument is the logical grouping of any of the VISA resources. The VISA Instrument Control Resources Organizer serves as a means to group any number of any type of VISA Instrument Control Resources within a VISA system.

VISA

Virtual Instrument Software Architecture. VISA is a standard I/O library that allows software from different vendors to run together on the same platform. Agilent VISA is part of the Agilent IO Libraries Suite.

VISA address

A resource descriptor that can be used to open a VISA session.

VISA alias

A string that can be used instead of a resource descriptor in VISA programs. Using VISA aliases rather than hard-coded resource descriptors makes your programs more portable. You can define VISA aliases for your instruments in Connection Expert.

VISA COM

The VXI*plug&play* specification for a COM-compliant VISA I/O library and its implementation. Agilent VISA COM is part of the Agilent IO Libraries Suite.

VISA Instrument Control Resources

The VISA definition of device-specific resource classes. VISA Instrument Control Resources include all VISA-defined device and interface capabilities for direct, low-level instrument control.

VISA name

The prefix of a VISA address, also called the **VISA interface ID**. The VISA name specifies the interface.

VISA resource manager

The part of VISA that manages resources. This management includes support for opening, closing, and finding resources, setting attributes, retrieving attributes, and generating events on resources.

VISA resource template

The part of VISA that defines the basic constraints and interface definition for the creation and use of a VISA resource. Each VISA resource must derive its interface from the VISA resource template.

VXI Resource Manager

A software utility that initializes and prepares a VXI system for use. The VXI Resource Manager is part of the Agilent IO Libraries Suite.

Windows notification area

See **notification area**.

Index

A

addressing device sessions, 34
addressing RS-232 devices, 147
addressing RS-232 interfaces, 152
addressing VXI message-based devices, 112
Agilent
  telephone numbers, 14
  web site, 14
Agilent 488, 207
asynchronous events, enabling/disabling, 57
asynchronous events, handling, 56

B

buffers, formatted I/O, 44, 52
building SICL applications, 30

C

C applications, compiling, 31
command module, 110
commander session, 33
common LAN problems, 201
communications sessions, opening, 32
compiled SCPI (C-SCPI), 110
compiling C applications, 31
configuring RS-232 interfaces, 141
Connection Expert, 18

D

device session, 32
device sessions, addressing, 34
device sessions, RS-232, 143
device types, VXI, 109
DLLs, C applications, 30

E

error handlers
  using in Visual Basic, 62
error handlers, using in C, 59
error handling, 59
Event Viewer, 59
Event Viewer utility, 196
examples
  C Example Program Code, 16
  Configuring RS-232 Interface, 141
  Creating a Commander Session, 36
  Device Locking (C), 66
  Device Locking (Visual Basic), 67
  Error Handlers (Visual Basic), 63
  Formatted I/O (Visual Basic), 51
  GPIB (82350) Interface, 83
  GPIB Device Session (C), 88
  GPIB Device Session (Visual Basic), 89
  GPIB Interface Session (C), 94
  GPIB Interface Session (Visual Basic), 95
  Installing an Error Handler (C), 60
  LAN-gatewayed Addressing, 166
  LAN-gatewayed Session (C), 170
  LAN-gatewayed Session (Visual Basic), 171
  Non-Formatted I/O (C), 54
  Non-Formatted I/O (Visual Basic), 55
  Opening a Device Session, 34
  Opening an Interface Session, 35
  Oscilloscope Program (C), 69
  Oscilloscope Program (Visual Basic), 75
  Processing VME Interrupts (C), 136
  RS-232 Device Session (Visual Basic), 150
  RS-232 Interface Session (C), 155
  RS-232 Interface Session (Visual Basic), 157
  Servicing Requests (C), 100
  Visual Basic Program Example Code, 22
  VME Interrupts (C), 129
  VXI Interface Session (C), 124
  VXI Interrupt Actions (C), 135
  VXI Memory I/O (C), 132
  VXI Message-Based Device Session (C), 113
  VXI Register-Based Programming (C), 121
  Writing an Error Handler (C), 61

F

formatted I/O
  buffers, 44, 52
  C applications, 37
  conversion, 38, 46
  related functions, 45
  Visual Basic applications, 45
  Visual Basic functions, 53

G

getting started using C, 16
getting started using Visual Basic, 22
GPIB
  handling SRQs, 99
  interface sessions, 92
  interrupt handlers, 98
  multiple interrupts, 99
  primary/secondary addresses, 87
  VXI mainframe connections, 87
GPIB commander sessions, 97
  interrupts, 98

GPIB communications sessions, selecting, 85
GPIB device sessions, service requests, 88
GPIB device sessions, SICL functions, 86
GPIB device sessions, using, 86
GPIB devices, addressing, 86
GPIB interface sessions
  service requests, 94
GPIB interface sessions, interrupts, 93
GPIB interface sessions, SICL functions, 92
GPIB Interfaces, configuring, 83
GPIB interfaces, introduction, 82
GPIB SICL functions, 85

H

handling errors, 59

I

I/O commands, sending, 36
interface session, 33
interface sessions, RS-232, 143, 152
interpreted SCPI (I-SCPI), 110
interrupt handlers, 57
interrupts, 56
I-SCPI interface, 115

L

LAN
  application terminations/timeouts, 181
  clients and threads, 163
  default timeout values, 178
  hardware architecture, 160
  interface sessions, 174
  interface sessions, SICL functions, 174
  interfaces overview, 160
  IP addresses, 166
  Lan-gatewayed sessions, 165
  locks, 176
  SICL configuration, 163
  SICL performance, 163
  threads, 176
  timeout functions, 178
  timeouts, 177
  timeouts in multi-threaded applications, 180
  Using the ping Utility, 201
LAN interface sessions, 174
  SICL functions, 174
LAN interfaces, overview, 160
LAN-gatewayed sessions, 165
libraries, C applications, 30
locking
  multi-user environment, 66
locks
  actions, 65
  locking multi-user environment, 66
locks, using, 64

M

message-based devices, programming, 111
message-based devices, VXI, 109

N

NI-488, 207
non-formatted I/O, 53

O

opening communications sessions, 32
overview, guide, 10
overview, SICL, 11

P

peeks and pokes, register, 110
programming VXI register-based devices, 115

R

register peeks and pokes, 110
register-based devices, 109
register-based devices, programming, 115
RS-232
  common problems, 200
  communications sessions, 142
  device sessions, 143, 147
  interface sessions, 143, 152
  interface sessions, SICL functions, 152
  SICL functions, 144
RS-232 device sessions
  SICL function support, 148
RS-232 devices, addressing, 147
RS-232 interfaces, addressing, 152
RS-232 interfaces, configuring, 141
RS-232 interfaces, introduction, 140

S

sample code
  See also examples, 9
selecting GPIB communications sessions, 85
sending I/O commands, 36
SICL
  description, 12
  GPIB functions, 85
  GPIB interface sessions, 92
  SICL declaration file, 30
SICL applications, building, 30
SICL error codes, 196
SICL error messages, logging, 59
SICL functions, GPIB device sessions, 86
SICL functions, VXI interfaces, 110
SICL overview, 11
SICL programs, troubleshooting, 196
SRQ handlers, 57
SRQs, 56
status byte, 88

T

troubleshooting
  common LAN problems, 201
  common Windows problems, 200
  LAN client problems, 202
  LAN server problems, 204
  RS-232 problems, 200
  SICL
    error codes, 196

SICL programs, 196

U

USB
communicating with instruments using SICL, 185
interfaces overview, 184
using GPIB commander sessions, 97
using GPIB interface sessions, 92
using RS-232 interface sessions, 152
using VXI interface sessions, 123

V

VISA, 11
Visual Basic applications, running, 32
VME devices
communicating with, 126
declaring resources, 126
interrupts, 129
mapping VME memory, 127
reading/writing to device registers, 128
unmapping memory space, 128
VXI
backplane memory I/O performance, 131
block memory access, 132
command module, 116
compiled SCPI, 116
device types, 109
I-SCPI interface, 115
message-based devices, 109
message-based devices, addressing, 112
programming message-based devices, 111
register programming, 115
register-based devices, VXI, 109
SICL function support, 131
single location peek/poke, 131
VXI interface sessions, 123
VXI interfaces, SICL functions, 110

W

Windows applications, thread support, 32

X

XON/XOFF, 153