Agilent E2094 SICL Libraries

Agilent SICL User's Guide for Windows

Agilent Technologies

Notices

Manual Part Number

E2094-91001

Edition

Sixth edition, April 2003

Printed in USA

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

Warranty

Technology Licenses

Restricted Rights Legend

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

This *Agilent Standard Instrument Control Libraries (SICL) User's Guide for Windows* describes Agilent SICL and how to use it to develop I/O applications on Microsoft Windows 98SE, Windows Me, Windows 2000, Windows XP, and Windows NT 4.0. 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 SICL on your computer. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for installation on Windows systems. Unless otherwise indicated, Windows NT refers to Windows NT 4.0.

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 an example 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 SICL error codes, as well as porting to Visual Basic and on RS-232 cables.
- *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. The Agilent IO Libraries consists of two libraries: *Agilent Virtual Instrument Software Architecture (VISA)* and *Agilent Standard Instrument Control Library (SICL)*.

Introducing VISA and SICL

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

NOTE

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

SICL Description

Agilent Standard Instrument Control Library (SICL) is an I/O library developed by 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

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

SICL Users

SICL is intended for instrument I/O and C/C++ or Visual Basic programmers who are familiar with Windows 98SE, Windows Me, Windows 2000, Windows XP, or Windows NT. To perform SICL installation and configuration on Windows 2000, Windows XP, or Windows NT, 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	Information is provided in the form of Windows Help.
SICL Example Programs	Example programs are provided online to help you develop SICL applications. If the default installation directory was used, SICL example programs are provided in the `C:\Program Files\Agilent\IO Libraries\c\samples\` subdirectory (for C/C++), and in the `C:\Program Files\Agilent\IO Libraries\vb\samples\` subdirectory (for Visual Basic).
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 and Canada, you can reach Agilent Technologies at these telephone numbers:

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

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

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

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 example programs in C/C++ and in Visual Basic to help you verify your configuration and introduce you to some of SICL's basic features. The chapter contents are:

- Getting Started Using C
- Getting Started Using Visual Basic

NOTE

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

You may want to review the SICL Language Reference in online Help to familiarize yourself with SICL functions. To see the reference information online, double-click the **Help** icon in the Agilent IO Libraries program group.

Getting Started Using C

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

C Example Program Code

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

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

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

```
/* This program uses the Standard Instrument
Control Library to query a GPIB instrument for
an identification string and then prints the
result. This program is to be built as a WIN32
console application on Windows 98SE, Windows Me,
Windows 2000, Windows XP, or Windows NT. 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
IO Config utility.
```

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

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

/* Install a default SICL error handler that
logs an error message and exits. On Windows 98SE
or Windows Me, view messages with the SICL
Message Viewer. For Windows 2000, XP, or NT, use
the Event Viewer. */

ionerror(I_ERROR_EXIT);

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

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

/* Write the *RST string (and send an EOI
   indicator) to put the instrument into a known
   state. */
iprintf(id, "*RST\n");
/* Write the *IDN? string and send an EOI
indicator, then read the response into buf.*/
ipromptf(id, "*IDN?\n", "%t", buf);

printf("%s\n", buf);
iclose(id);

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

}
```

C Example Code Description

sicl.h

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

INST

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

ionerror

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

NOTE

On Windows 98SE and Windows Me, error messages may be viewed by executing the **Message Viewer** utility in the Agilent IO Libraries program group. On Windows 2000, XP, and NT, these messages may be viewed with the **Event Viewer** utility in the Agilent IO Libraries 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 IO Config utility. The bus (primary) address of the instrument follows (**0** in this case) and is typically set with switches on the instrument or from the front panel of the instrument.

NOTE

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

itimeout

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

iprintf and ipromptf

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

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

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

iclose and _siclcleanup

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

Compiling the C Example Program

The `c\samples\idn` subdirectory (default path `C:\Program Files\Agilent\IO Libraries\c\samples\idn`) contains a number of files you can use to build the example with specific compilers. You will have a subset of the following files, depending on the Windows environment you are using.

Table 2 File Listing for c\samples\idn subdirectory

idn.c	Example program source file.
idn.def	Module definition file for IDN example program.
MSCIDN.MAK	Windows 3.1 makefile for Microsoft C and Microsoft SDK compilers.
VCIDN.MAK	Windows 3.1 project file for Microsoft Visual C++.
VCIDN32.MAK	Windows 98SE/Me /2000/XP/NT (32-bit) project file for Microsoft Visual C++.
BCIDN32.IDE	Windows 98SE/Me/2000/XP/NT (32-bit) project file for Borland C Integrated Development Environment.

Steps required to compile the **idn** example program follow.

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

2 Change directories to the location of the example.

3 The program assumes the GPIB interface name is **gpib0** (set using IO Config) and the instrument is at bus address **0**. If necessary, modify the interface name and instrument address on the DEVICE_ADDRESS definition line in the *IDN.C* source file.

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

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

Running the C Example Program

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

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

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

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

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

Where to Go Next

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

- *Chapter 4 - Using SICL with GPIB*
- *Chapter 5 - Using SICL with 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).

Porting to Visual Basic 6.0

This edition of this manual shows how to program SICL applications in Visual Basic version 6.0. For SICL applications written in an earlier Visual Basic version than 6.0 (for example, version 3.0), you can port your SICL applications to Visual Basic version 6.0. Once you have made the changes shown, your SICL applications should run correctly with Visual Basic 6.0.

To port SICL applications to Visual Basic 6.0, you will need to add the *SICL32.BAS* declaration file (rather than the *SICL.BAS* file) as a module to each project that calls SICL for Visual Basic 6.0.

There may also be changes in functions when passing null pointers for strings to SICL functions. For example, in Visual Basic version 3.0, the preceding **ByVal** keyword was used as **`ivprintf`(*id*, *mystring*, ByVal 0&)**. However, in Visual Basic version 4.0 or later, you only need to pass the **0&** null pointer because version 4.0 or later knows this is by reference (**ivprintf(*id*, *mystring*, 0&)**).

In Visual Basic 6.0, project files have a **.vbp** extension, not a **.mak** extension. Earlier versions that used a **.mak** project suffix may be imported into VB 6.0 by selecting **Open Project...** and choosing a project with a **.mak** extension from an earlier version of Visual Basic. When you save the project, VB 6.0 will append a **.vbp** to the project file.

Constants in Visual Basic 3.0 and 4.0, such as MB_ICON_EXCLAMATION and MB_OK are not defined in Visual Basic 6.0. Instead, use constants such as **`vbExclamation`** and **`vbOK`**.

Print statements should be changed to **Debug.Print** or **Form1.Print**. Output will then be directed to print to the Immediate window or to a Form named **Form1**, respectively. Otherwise, you will get an error: **Method not valid without suitable object**.

Visual Basic Program Example Code

This section describes an example program called **idn** that queries a GPIB instrument for its identification string. This example builds a console application for WIN32 programs (32-bit SICL programs on Windows 98SE, Windows Me, Windows 2000, Windows XP, or Windows NT) using the Microsoft Visual Basic 6.0 Programming language.

NOTE Be sure to include the *SICL32.BAS* file (in the VB directory) in your Visual Basic project. This file contains the necessary SICL definitions, function prototypes, and support procedures to allow you to call SICL functions from Visual Basic.

All files used to develop SICL applications in Visual Basic 6.0 are located in the vb subdirectory of the base IO Libraries directory. The default install location of the IO Libraries directory is `C:\Program Files\Agilent`.

Sample Visual Basic programs are located in the `C:\Program Files\Agilent\IO Libraries\vb\samples` subdirectory. Each sample program subdirectory contains a (.vbp) project file that you can open from Visual Basic 6.0.

The **idn** example files are located in the `vb\samples\idn` subdirectory under the base IO Libraries 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 example 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:
  ' Start|Programs|Agilent IO Libraries|IO
  ` Config
  ' "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)

  ' Tell SICL to clean up for this task
  Call siclcleanup

End Sub
```

Visual Basic Example 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 IO Config 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

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

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 and _siclcleanup

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

Building and Running the VB Example Program

The `vb\samples\idn` subdirectory contains the files you can use to build and run the example. You will have a subset of the following files, depending on the Windows environment you are using.

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** example 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 10. Otherwise, go to Step 3.

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**. From the menu, select **Project | Add Module**, select the **Existing** tab, and browse to the idn directory.

4 The **idn** example files are located in directory `C:\Program Files\Agilent\IO Libraries\vb\samples\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, **Form1** may be deleted if desired. 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.

7 To add this module to your project, from the menu select **Project | Add Module**, select the **Existing** tab, browse to the `vb\` directory under the IO Libraries install directory, select *sicl32.bas*, and click **Open**.

8 At this point, the Visual Basic project can be run and debugged. You will need to edit the *idn.bas* module code to change the SICL Interface Name and address in the code to match your device configuration.

9 The program assumes the SICL interface name is **gpib0** (set using IO Config) and the instrument is at bus address **22**. If necessary, modify the interface name and instrument address.

10 If the program runs correctly, an example of the output if connected to a Hewlett-Packard 34401A Multimeter would be:

```
AGILENT,34401A,0,4-1-1
```

11 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.

12 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 correcting the problem.

Where to Go Next

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

- *Chapter 4 - Using SICL with GPIB*
- *Chapter 5 - Using SICL with 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. Example programs are provided to help you develop SICL applications.

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

For C/C++: `C:\Program Files\Agilent\IO Libraries\c\samples\`

For Visual Basic: `C:\Program Files\Agilent\IO Libraries\vb\samples\`

The chapter includes:

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

NOTE For details about SICL functions, see 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 3.0 or earlier programs, you must add the *SICL.BAS* file to each project that calls SICL. For Visual Basic version 4.0 or later programs, you must add the *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. (Borland compilers use *BCSICL32.DLL*.)

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

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

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

Compiling and Linking C Applications

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

NOTE

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

Microsoft Visual C++ Compilers

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

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

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

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

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

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

Borland C++ Version 4.0 Compilers

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

2 Edit the *BCC32.CFG* and *TLINK32.CFG* files, located in the `bin` subdirectory of the Borland C installation directory.

   a Add the following line to *BCC32.CFG* so the compiler can find the *sicl.h* file: `-IC: \IO_base_dir\c`, where `IO_base_dir` is the IO Libraries base directory.

   b Add the following line to both files so the compiler and linker can find *BCSICL32.LIB*: `-LC: \IO_base_dir\c`, where `IO_base_dir` is the IO Libraries base directory.

   c For example, to create *MYPROG.exe* from *MYPROG.C*, type: `BCC32 MYPROG.C BCSICL32.LIB`.

Loading and Running Visual Basic Applications

To load and run an existing Visual Basic application, first run Visual Basic. Then, open the project file for the program you want to run by selecting **File | Open Project** from the Visual Basic menu. Visual Basic project files have a *.MAK* file extension. After you 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 stand-alone (just like any other *.exe* file) without having to run Visual Basic.

Thread Support for 32-bit Windows Applications

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

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 the variable is declared, you can open the communication channel by using the SICL **iopen** function, as shown in the following example.

C example:

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

Visual Basic example:

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

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

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

Device Sessions

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

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

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

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

Table 3 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 symbolic name are set by running the IO Config utility from the Agilent IO Libraries Control (**IO** icon on the taskbar) for Windows 98SE, Windows Me, Windows 2000, Windows XP, or Windows NT. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on the IO Config utility.

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

C example:

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

Visual Basic example:

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

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

Table 4 Valid Addresses for Interfaces

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

Examples: Opening an Interface Session These examples open an interface session with an RS-232 interface.

C example:

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

Visual Basic example:

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

Commander Sessions

A **commander session** allows your computer to talk to the interface controller. Typically, the controller is the computer used to communicate with devices on the interface. When the computer is not 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 address followed by a comma, and then the string **cmdr** in the **iopen** function. The following are valid commander addresses.

Table 5 Valid Commander Addresses

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

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

C example:

```
INST cmdr;
cmdr = iopen(“gpib0,cmdr”);
```

Visual Basic example:

```
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 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 are:

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

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

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

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

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

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

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

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

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

```
%[format flags][field width][. precision]
[, array size][argument modifier]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 6 Formatting 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 padded character is dependent on various flags. You can use an asterisk (*) in place of the integer to indicate that the integer is taken from the next argument.

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

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

Pads to six characters: **61**

. Precision 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 7 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 8 iprintf and ipromptf Format Codes in C Applications

Format Codes	Description
d,i	Corresponding argument is an integer.
f	Corresponding argument is a float.

Table 8 iprintf and ipromptf Format Codes in C Applications

Format Codes	Description
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 9 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.

Table 9 iscanf and ipromptf Format Codes

Format Codes	Description
o,u,x	Corresponding argument must be a pointer to an unsigned integer.
[	Corresponding argument must be a character pointer.
F	Corresponding argument is a pointer to a FILE descriptor opened for writing.

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

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

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

#if defined(__BORLANDC__) && !defined(__WIN32__)
  _InitEasyWin(); /*Required for Borland EasyWin
                    programs*/
#endif
/* Log message and terminate on error */
ionerror (I_ERROR_EXIT);

/* Open the multimeter session */
dvm = iopen (“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);

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

  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 at what point you want the data written to the device.

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

This can be very useful in queuing up data to send to a device. It can also raise I/O performance by doing a few large writes instead of several smaller writes. This behavior can be changed by the **isetbuf** and **isetubuf** functions. See “Formatted I/O Buffers” in the following section for details.

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

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

The write buffer is maintained by the **iprintf** and the write portion of the **ipromptf** functions. It queues characters to send to the device so that they are sent in large blocks, thus increasing performance. The write buffer automatically flushes when it sends a newline character from the format string (see the **%t** 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 10 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 11 Format Flags for ivprintf in Visual Basic

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

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

```
Dim numb As Integer
Dim ret_val As Integer
```

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

Sends: **61.000000**

Field Width 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 12 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.

s
Table 13 ivprintf format codes in Visual Basic Application

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

s
Table 14 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 on Visual Basic.

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

```
Dim ret_val As Integer
Dim data As String * 180
ret_val = ivscanf(id, “%180s”, data)
```

```
‘Example: Formatted I/O (Visual Basic)

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:
  'Start | Programs | Agilent IO Libraries | IO
  ` Config
  '"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)

  'Tell SICL to cleanup for this task
  Call siclcleanup

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

Table 15 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 example:

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

VB example:

```
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 example:

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

VB example:

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

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

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

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

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

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

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

  /* Open the multimeter session */
  dvm = iopen (“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);

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

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

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

  ' Open the multimeter session
  dvm = iopen(“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)

  ' Tell SICL to clean up for this task
  Call siclcleanup

  Exit Sub

End Sub
```

Handling Asynchronous Events

Asynchronous events are events that happen outside the control of your application. These events include Service ReQuests (**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 performs an **ExitProcess** on Windows 98SE or Me, or calls **iclose** on Windows 2000, XP, or NT. 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 98SE, Windows Me, Windows 2000, Windows XP, and Windows NT, SRQ and interrupt handlers execute on a separate thread (a thread created and managed by SICL). This means a handler can be executing when the **iintroff** call is made. If this occurs, the handler will continue to execute until it has completed.

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

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

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

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

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

```
...
ionintr (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** (Windows 2000, XP, and NT) or the **Message Viewer** (Windows 98SE and Me) to log SICL error messages.

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

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

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

Using Error Handlers in C

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

Installing an error handler allows your application to ignore the return value, and permits the error procedure to detect errors and recover. The error handler is called before the function that generated the error completes. 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 16 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_NOEXIT	This value installs a special error handler which will log a diagnostic message and then allow the process to continue execution.

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

Example: Installng an Error Handler (C) Typically, error handling code is intermixed with the I/O code in an application. However, with SICL error handling routines no special error handling code is inserted between the I/O calls. Instead, a single line at the top (calling **ionerror**) installs an error handler that gets called any time an error occurs. In this example, a standard, system-defined error handler is installed that logs a diagnostic message and then exits.

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

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

main ()
  {
  INST dvm;
  double res;

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

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

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

  return 0;
  }
```

Example: Writing an Error Handler (C) This is an example of writing and implementing your own error handler.

NOTE

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

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

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

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

  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);

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

Using Error Handlers in Visual Basic

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

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

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

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

```
On Error GoTo MyErrorHandler
```

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

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

```
On Error Resume Next
```

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

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

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

```
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:

  'Start | Programs | Agilent IO Libraries | IO
  ` Config
  '"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

  'Tell SICL to cleanup for this task
  Call siclcleanup

End Sub
```

Using Locks

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

What are Locks?

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

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

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

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

Not all SICL routines are affected by locks. Some routines that set or return session parameters never touch the interface hardware and therefore work without locks. For information on using locks in multi-threaded SICL applications over LAN, see *Chapter 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 the "Using Locks" section, an interface session can access a device locked from a device session.

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

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

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

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

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

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

  return 0;}
```

Example: 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)

  'Tell SICL to cleanup for this task
   Call siclcleanup

   Exit Sub
```

```
ErrorHandler:

   'Display the error message.
   MsgBox "*** Error : " + Error
   'Tell SICL to cleanup for this task
   Call siclcleanup

End Sub
```

Additional Example Programs

This section contains two additional example programs that provide guidelines to help you develop SICL applications, including Example: Oscilloscope Program (C) and Example: Oscilloscope Program (Visual Basic).

Example: Oscillosope Program (C)

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

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

Table 17 Program Files for the C Oscilloscope Program

SCOPE.C	Example program source file.
SCOPE.H	Example program header file.
SCOPE.RC	Example program resource file.
SCOPE.DEF	Example program module definitions file.
SCOPE.ICO	Example program icon file.

Table 17 Program Files for the C Oscilloscope Program

VCSCP32.MAK	Windows 98SE/Me/2000/XP/NT project file for Microsoft Visual C++.
BCSCP32.IDE	Windows 98SE/Me/2000/XP/NT project file for Borland C Integrated Development Environment.

Building the Project File This section shows how to create the project file for this example using Microsoft Visual C. You can also load the makefile directly from the `C:\Program Files\Agilent\IO Libraries\c\samples\scope` subdirectory, if you desire. If you are using another language tool, choose the appropriate project file or makefile from the `c\samples\scope` subdirectory.

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

1 Select **File | New** from the menu and select **Project** from the list box that appears. Then click **OK**.

2 The **New Project** dialog box is displayed. Type the name you want for the project in the edit box labeled **Project Name**. Then, select **Application** from the **Project Type** list box. Select the directory location for the project in the **Directory** list box and click the **Create** button.

3 The **Project Files** dialog box is now displayed. Double-click the source files *scope.c*, *scope.rc*, and *scope.def* to add them to the project. Also add *sicl32.lib* from the SICL `C` directory. Then, click the **Close** button.

4 Select **Project | Settings** from the menu and click the **C\C++** button. Select **Code Generation** from the **Category** list box. Then, select **Multithreaded Using DLL** from the **Use Run-Time Library** list box and click **OK**.

5 Select **Tools | Options** from the menu and click the **Directories** button in the **Options** dialog box. Select **Include Files** from the **Show Directories for:** list box and click the **Add** button. Then, type `\SICL\C` and click **OK**.

6 Select **Project | Build** to build the application.

If there are no errors reported, you can execute the program by selecting **Project | Execute**. An application window will open. Several commands are available from the **Actions** menu, and any results or output will be printed in the program window. To end the program, select **File | Exit** from the program menu.

Program Overview 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 example program uses a few of the capabilities of the **iprintf/iscanf/ipromptf** functions and their derivatives.

The **iprintf** function is used to send commands. As with all of the formatted I/O functions, the data is actually buffered. In this call, the **\n** at the end of the format:

```
iprintf(id,":waveform:preamble?\n");
```

causes the buffer to be flushed and the string to be output. If desired, several commands can be formatted before being sent and then all commands outputted at once. The formatted I/O buffers are automatically flushed whenever the buffer fills (see **isetbuf**) or when an **iflush** call is made.

When reading data back from a device, the **iscanf** function is used. To read the preamble information from the oscilloscope, use the format string "**%,20f\n**":

```
iscanf(id,"%,20f\n",pre);
```

This string expects to input 20 comma-separated floating point numbers into the **pre** array.

To upload the oscilloscope waveform data, use the string "**%#wb\n**". The **wb** indicates that **iscanf** should read word-wide binary data. The **#** preceding the data modifer tells **iscanf** to get the maximum number of binary words to read from the next parameter (**&elements**):

```
iscanf(id,"%#wb\n",&elements,readings);
```

The read will continue until an EOI indicator is received or the maximum number of words have been read.

Interface Sessions Sometimes it may be necessary to control the GPIB bus directly instead of using SICL commands. This is accomplished using an interface session and interface-specific commands. This example uses **igetintfsess** to get a session for the interface to which the oscilloscope is connected. (If you know which interface is being used, it is also possible to just use an **iopen** call on that interface.)

Then, **igpibsendcmd** is used to send some specific command bytes on the bus to tell the printer to listen and the oscilloscope to send its data. The **igpibatnctl** function directly controls the state of the ATN signal on the bus.

```
void print_disp (INST id)
  {
  INST 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 example handler, the oscilloscope status is read to verify that the oscilloscope asserted SRQ, and then the SRQ is cleared and a status message is displayed. If the oscilloscope did not assert SRQ, the handler prints an error message.

```
void SICLCALLBACK my_srq_handler(INST id)
  {
  unsigned char status;

  /* make sure it was the scope requesting
     service */
  ireadstb(id,&status);

  if (status &= 64) {
    /* clear the status byte so the scope can
       assert SRQ again if needed. */
    iprintf(id,"*CLS\n");
    sprintf(text_buf[num_lines++], "id = %d, SRQ
      received!, stat=0x%x", id,status);
  } else {
    sprintf(text_buf[num_lines++],
      "SRQ received, but not from the scope");
    }
  InvalidateRect(hWnd, NULL, TRUE);
  }
```

In the routine that commands the oscilloscope to print its display, the oscilloscope is set to assert SRQ when printing is finished. While the oscilloscope is printing, the example program has the application suspend execution. SICL provides the function **iwaithndlr** that will suspend execution and wait until either an event occurs that would call a handler, or a specified timeout value is reached.

In the example, interrupt events are turned off with **iintroff** so that all interrupts are disabled while interrupts are being set up. Then, the SRQ handler is installed with **ionsrq**. Code to program the oscilloscope to print and send an SRQ is next, then the call to **iwaithdlr**, with a timeout value of 30 seconds. When the oscilloscope finishes printing and sends the SRQ, the SRQ handler will be executed and then **iwaithdlr** will return. A call to **iintron** re-enables interrupt events.

```
void print_disp (INST id)
  {
  ...

  iintroff();
  ionsrq(id,my_srq_handler);/* Not supported on
                               82335 */

  /* tell the scope to SRQ on 'operation
     complete' */
  iprintf(id,"*CLS\n");
  iprintf(id,"*SRE 32 ; *ESE 1\n") ;

  /* tell the scope to print */
  iprintf(id,":print ; *OPC\n") ;

  ... code to tell the scope to print
  /* wait for SRQ before continuing program */
  iwaithdlr(30000L);
  iintron();

  sprintf (text_buf[num_lines++],"Printing
    complete!") ;
  ...
  }
```

Example: Oscillosope Program (Visual Basic)

This Visual Basic example program uses SICL to get and plot waveform data from an Agilent 54601A (or compatible) oscilloscope. This routine is called each time the **cmdGetWaveform** command button is clicked.

Program Files The oscilloscope example files are located in the `C:\Program Files\Agilent\IO Libraries\vb\samples\scope` subdirectory, if IO Libraries was installed to the default directory. The files are listed in the following table.

Table 18 Files Used for the Oscilloscope Example Program

SCOPE.FRM	Visual Basic source for the SCOPE example program.
SCOPE.MAK	Visual Basic project file for the SCOPE example program.

Loading and Running the Program Follow these steps to load and run the **SCOPE** sample program:

1 Connect an Agilent 54601A oscilloscope to your interface.

2 Run Visual Basic 6.0.

3 Open the project file *scope.vbp* by selecting **File | Open Project** from the Visual Basic menu.

4 The SICL Visual Basic declaration file *sicl32.bas* module must be added to your VB project. To add this module to your project, from the menu select **Project | Add Module**, select the **Existing** tab, browse to the `vb\` directory under the IO Libraries install directory, select *sicl32.bas*, and click **Open**.

5 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
```

6 Run the program by pressing the **F5** key or the by clicking the **RUN** button on the Visual Basic Toolbar.

7 Press the **Waveform** button to get and display the waveform.

8 Press the **Integral** button to calculate and display the integral.

9 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 19 Functions of the Example Program

Listing	Description
CmdGetWaveform_Click	Subroutine that is called when the **cmdGetWaveform** command button is pressed. The command button is labeled **Waveform**.

Table 19 Functions of the Example Program

Listing	Description
On Error	This Visual Basic statement enables an error handling routine within a procedure. In this example, an error handler is installed starting at label **ErrorHandler** within the **cmdOutputCmd_Click** subroutine. The error handling routine is called any time an error occurs during the processing of the **cmdGetWaveform_Click** procedure. SICL errors are handled in the same way that Visual Basic errors are handled with the **On Error** statement.
CmdGetWaveform.Enabled	The button that causes the **cmdGetWaveform_Click** routine to be called is disabled when code is executing inside **cmdOutputCmd_Click**. This is good programming style.
iopen	An **iopen** call is made to open a device session for the oscilloscope. The device address for the oscilloscope is in the **scope_address** string.In this example, the default address is "*gpib0,7*." The interface name **gpib0** is the name given to the interface with the IO Config utility. The bus (primary) address of the oscilloscope follows, in this case 7. You may want to change the **scope_address** string to specify the correct address for your configuration.
igetintfsess	**igetintfsess** is called to return an interface session *id* for the interface to which the oscilloscope instrument is connected. This interface session will be used by the following **iclear** call to send an interface clear to reset theinterface.
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.

Table 19 Functions of the Example Program

Listing	Description
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.
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.

Table 19 Functions of the Example Program

Listing	Description
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**.
CmdGetWaveform.Enabled	This re-enables the button that causes the **cmdGetWaveform_Click** routine to be called. This allows the program to get another waveform.
Exit Sub	This Visual Basic statement causes the **cmdGetWaveform_Click** subroutine to be exited after processing an error in the subroutine's error handler.

4
Using SICL with GPIB

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

For C/C++: `C:\Program Files\Agilent\IO Libraries\c\samples\`

For Visual Basic: `C:\Program Files\Agilent\IO Libraries\vb\samples\`

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:

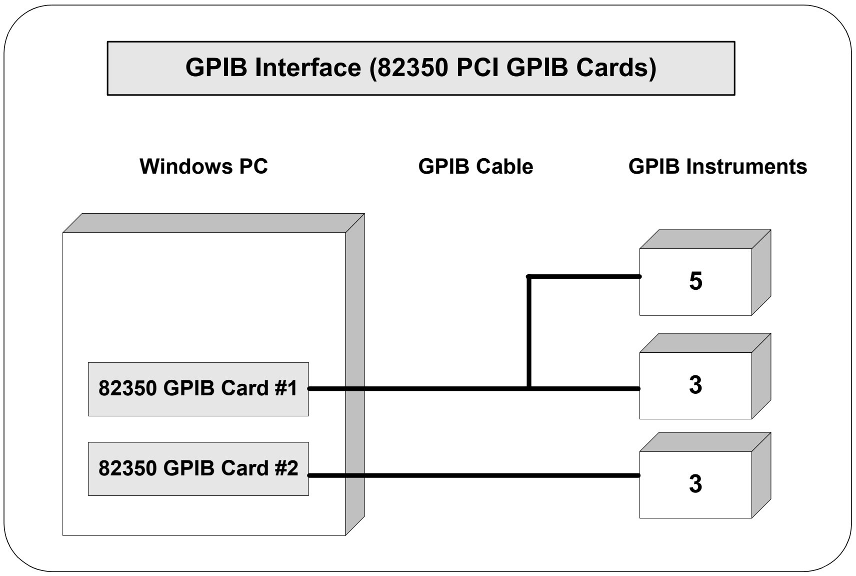
- GPIB Interfaces Overview
- Selecting a GPIB Communications Session
- SICL GPIB Functions

GPIB Interfaces Overview

This section provides an overview of GPIB interfaces, including typical hardware configuration, using IO Config, 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.



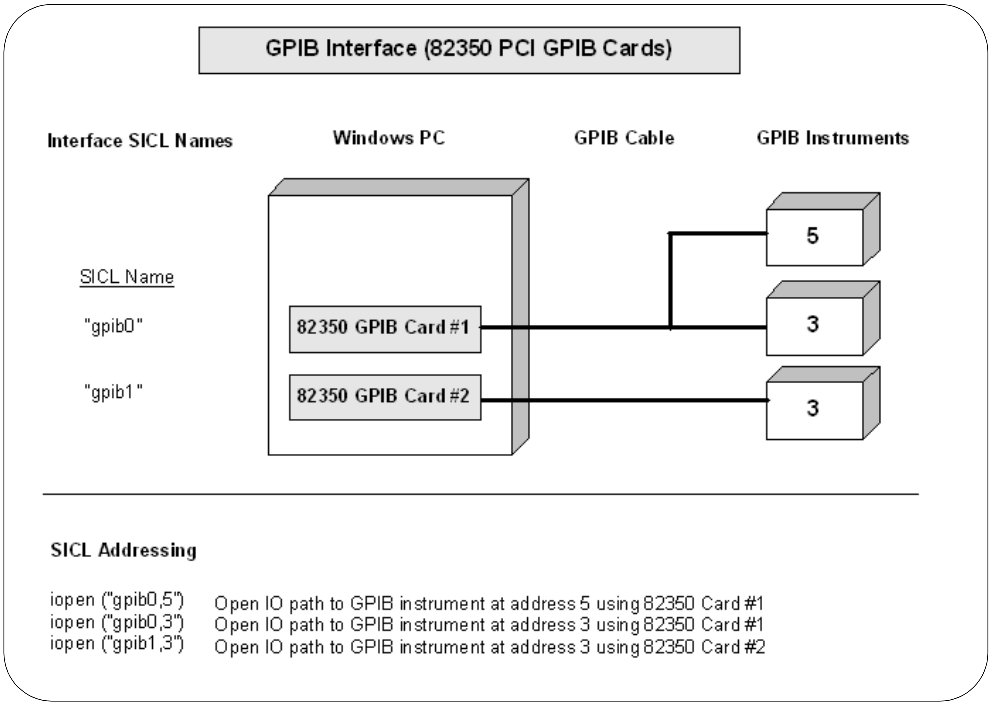
Configuring GPIB Interfaces

An **IO interface** can be defined as both a hardware interface and as a software interface. The purpose of the IO Config utility is to associate a unique interface name with a hardware interface.

The IO Libraries use 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. IO Config assigns an Interface Name and Logical Unit Number to the interface hardware, as well as other necessary configuration values for an interface when the interface is configured. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on IO Config.

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 IO Config 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 IO Config, you can use the **iopen** command to open the I/O paths shown.



```
iopen ("gpib0,5")   Open IO path to GPIB instrument at address 5 using 82350 Card #1
iopen ("gpib0,3")   Open IO path to GPIB instrument at address 3 using 82350 Card #1
iopen ("gpib1,3")   Open IO path to GPIB instrument at address 3 using 82350 Card #2
```

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 20 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 21 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 IO Config utility.

Opening IO Config To open IO Config, open the Agilent IO Libraries Control (**IO** icon on the taskbar) and click **Run IO Config**. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on the IO Config utility.

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

Table 22 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 Examples” for an example program to communicate with a VXI mainframe via the GPIB interface.

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

C example:

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

Visual Basic example:

```
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 Examples

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

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

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

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

main()
  {
  INST dvm;
  INST sw;
  double res;
  int i;
```

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

/* 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);

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

```
  return 0;
  }
```

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

```
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: Start | Programs | Agilent IO Libraries |
  'IO Config
  '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 23 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 IO Config utility.

Opening IO Config To open IO Config, open the Agilent IO Libraries Control (on the taskbar) and click **Run IO Config**. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on the IO Config utility. Example interface addresses follow.

Table 24 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 examples open an interface session with the GPIB interface.

C example:

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

Visual Basic example:

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

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 Examples

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

Example: 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 */

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

  /* exit process if SICL error detected */
  ionerror(I_ERROR_EXIT);
  /* open GPIB interface session */
  id = iopen(“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;
  }
```

Example: 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.

SICL Functions for GPIB Commander Sessions

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

Table 25 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 IO Config utility. To open IO Config, open the Agilent IO Libraries Control (**IO** icon on the taskbar) and click **Run IO Config**.

See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on the IO Config utility. Example GPIB addresses for commander sessions follow.

Table 26 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 examples open a commander session with the GPIB interface.

C example:

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

Visual Basic example:

```
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
}
```

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

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

Even if the different device sessions are in different processes, it is still important to stay in the SRQ handler until the SRQ line is released. However, it is not likely that a process 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 example programs in this chapter can be found in the following locations, if the Agilent IO Libraries were installed in the default directory:

For C/C++: `C:\Program Files\Agilent\IO Libraries\c\samples\`

For Visual Basic: `C:\Program Files\Agilent\IO Libraries\vb\samples\`

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

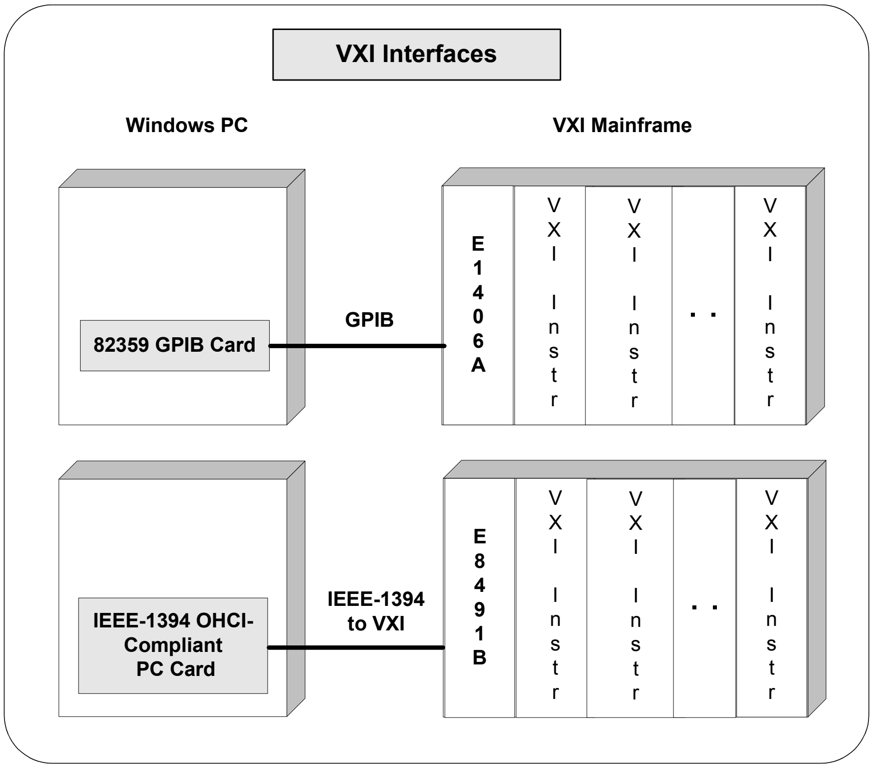
VXI Interfaces Overview

This section provides an overview of VXI interfaces, including typical hardware configuration, using IO Config, 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

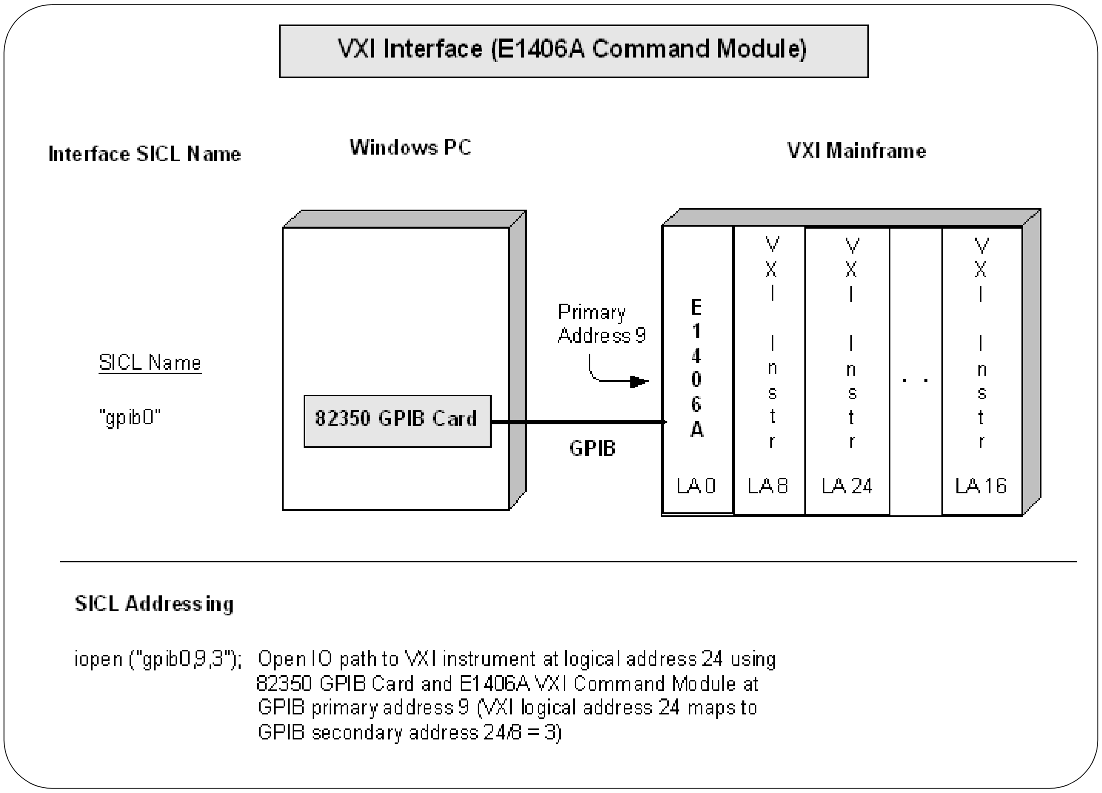
An **IO interface** can be defined as both a hardware interface and as a software interface. The purpose of the IO Config utility is to associate a unique interface name with a hardware interface.

The IO Libraries use 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. IO Config assigns an Interface Name and Logical Unit Number to the interface hardware, as well as other necessary configuration values for an interface when the interface is configured. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on IO Config.

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.

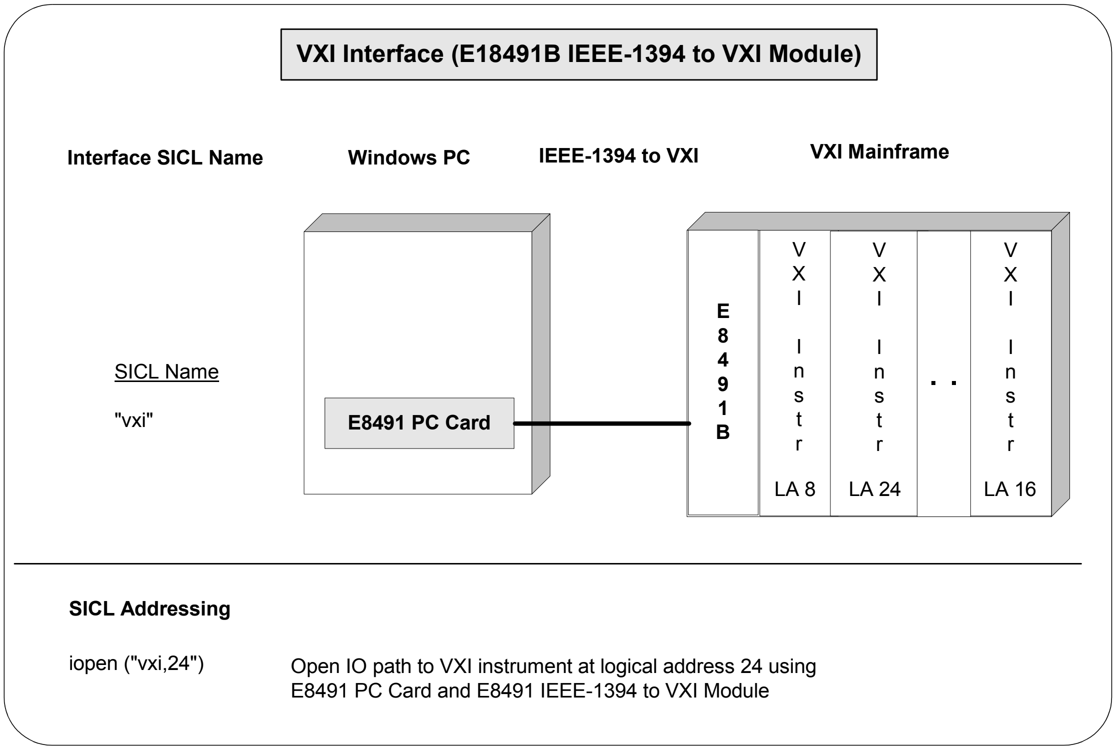
The IO Config utility has been used to assign the 82350 GPIB card a SICL name of **gpib0**. With these names assigned to the interfaces, the SICL addressing is as shown in the figure. For information on the E1406A Command Module, see the *Agilent E1406A Command Module User's Guide*. For information on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


VXI Interface (E1406A Command Module)
Interface SICL Name
Windows PC
VXI Mainframe
SICL Name
"gpib0"
82350 GPIB Card
Primary Address 9
GPIB
E1406A
VXI Instr
LA 0
LA 8
LA 24
LA 16
SICL Addressing
iopen ("gpib0,9,3"); Open IO path to VXI instrument at logical address 24 using 82350 GPIB Card and E1406A VXI Command Module at GPIB primary address 9 (VXI logical address 24 maps to GPIB secondary address 24/8 = 3)

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 IO Config 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*.



SICL Addressing

iopen ("vxi,24")	Open IO path to VXI instrument at logical address 24 using E8491 PC Card and E8491 IEEE-1394 to VXI Module

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 27 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 28 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 IO Config utility. To open IO Config, click the Agilent IO Libraries Control and then click **Run IO Config**. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on IO Config.

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 29 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”);
```

Example: VXI Message-Based Device Session (C)

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

```
/* vximdev.c
This example program measures AC voltage on a
multimeter 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);

  }
```

Example: 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:
  'Start | Programs | Agilent IO Libraries | IO
  ‘ Config
  '"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.

- **iI-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 IO Config utility. To open IO Config, click the Agilent IO Libraries Control **IO** icon on the taskbar and then click **Run IO Config**. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on IO Config.

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 30 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
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 31 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.

Table 31 Addressing Guidelines

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 Using the I-SCPI Interface

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

Using the I-SCPI Interface

To use the **iscpi** interface, you must first have configured the system with the IO Config utility to include **iscpi** as an interface. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on IO Config. When opening the device session, you will need to specify **iscpi** as the interface type in the SICL **iopen** call.

The **iscpi** interface includes drivers for most Agilent register-based devices. These drivers are located in the VISA directory specified during the Agilent IO Libraries installation (default is `C:\Program Files\VISA\Win95\bin` (Windows 98SE/Me) or `C:\Program Files\VISA\Winnt\bin` (Windows 2000/XP/NT). See the `C:\Program Files\VISA\Winxx\bin\iscpinfo.txt` file for a list of currently supported register-based devices.

I-SCPI SICL Functions

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

Table 32 SCPI SICL Functions

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

Addressing Guidelines

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

The simplest way to address a register-based device using the Interpreted SCPI (I-SCPI or **iscpi)** interface is to specify the interface logical unit or symbolic name and a device logical address in the *addr* parameter of the **iopen** function. I-SCPI automatically configures your system according to combining rules that determine how instruments are set up relative to other VXI instruments. For example:

```
dmm=iopen ("iscpi,24");
```

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

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

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

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

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

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

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

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

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

I-SCPI Interrupts and Service Requests

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

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

The interface cannot distinguish which device requested service, so **iscpi** acts as if all devices require service. Your SRQ handler can retrieve the device's **status byte** by using the **ireadstb** function. The status byte can be used to determine if the instrument needs service.

It is good practice to ensure that a device is not requesting service before leaving the SRQ handler. The easiest technique for this is to service all devices from one handler.

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

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

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

Example: I-SCPI Interface Session

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

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

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

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

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

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

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

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

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

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

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

Programming Directly to 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 33 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 34 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.

Example: VXI Register-Based Programming (C)

This example program opens a communication session with a register-based device connected to the address entered by the user. The program then reads the **Id** and **Device Type** registers and 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 35 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 IO Config utility. To open IO Config, click the Agilent IO Libraries Control **IO** icon on the taskbar and then click **Run IO Config**. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on IO Config.

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 36 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”);
```

Example: VXI Interface Session (C)

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

```
/* vxiintr.c
The following example gets information about a
specific vxi device and prints it out. */
```

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

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

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

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

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

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

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

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

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 Hewlett-Packard controllers.

Table 37 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.

Example: VME Interrupts (C)

This ANSI C example program opens a VXI interface session and sets up an interrupt handler. When the I_INTR_VME_IRQ1 interrupt occurs, the function defined in the interrupt handler 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 example program also requires the VME device to be using I_INTR_VME_IRQ1, and the controller to be the handler for the VME IRQ1.

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

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

#define ADDR "vxi"
```

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

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

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

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

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

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

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

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

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

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

  /* wait for interrupt */
  iwaithdlr (10000);
  /* turn interrupt notification on */
  iintron ();

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

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

VXI Backplane Memory I/O Performance

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

Table 38 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.

Example: VXI Memory I/O (C)

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

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

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

#define VXI_INST "vxi,24"

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

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

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

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

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

  data = iwpeek(addr) & 0xff;

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

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

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

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

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

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

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

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


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

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

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

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

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

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

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

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

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

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

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

Using VXI-Specific Interrupts

Example: VXI Interrupt Actions (C)

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

```
VME Interrupt arrives:
get iack value

send I_INTR_VME_IRQ?

is VME IRQ line configured VME only

if yes then
  exit
do lower 8 bits match logical address of one of
our servants?
if yes then
  /*  iack is from one of our servants */
  call servant_signal_processing(iack)
else
  /* iack is from non-servant VXI or VME device*/
  send I_INTR_VXI_VME interrupt to interface
  sessions

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

Example: Processing VME Interrupts (C)

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

#include <sicl.h>

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

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

  iintron ();
  iclose (id_intf1);
  }
  static void int_setup(INST id, unsigned long
    mask) {
  ionintr(id, int_hndlr);
  isetintr(id, I_INTR_VXI_SIGNAL, mask);
  }
  static void vmeint (INST id, unsigned short
    laddr) {
  int reg;
  char *a16_ptr = 0;

  reg = 8;
  a16_ptr = imap (id, I_MAP_A16, 0, 1, 0);
```

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

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

For C/C++: `C:\Program Files\Agilent\IO Libraries\c\samples\`

For Visual Basic: `C:\Program Files\Agilent\IO Libraries\vb\samples\`

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:

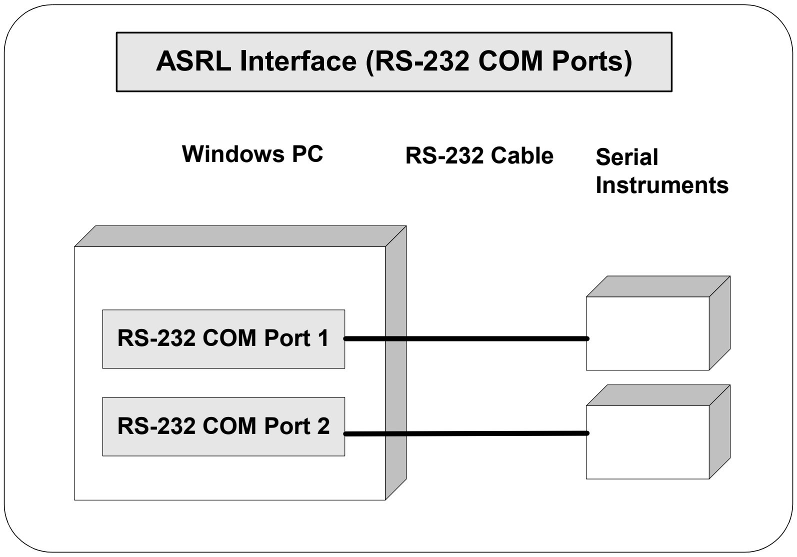
- ASRL (RS-232) Interfaces Overview
- Selecting an RS-232 Communications Session
- RS-232 SICL Functions

ASRL (RS-232) Interface Overview

This section provides an overview of RS-232 interfaces, including typical hardware configuration, using IO Config, 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.



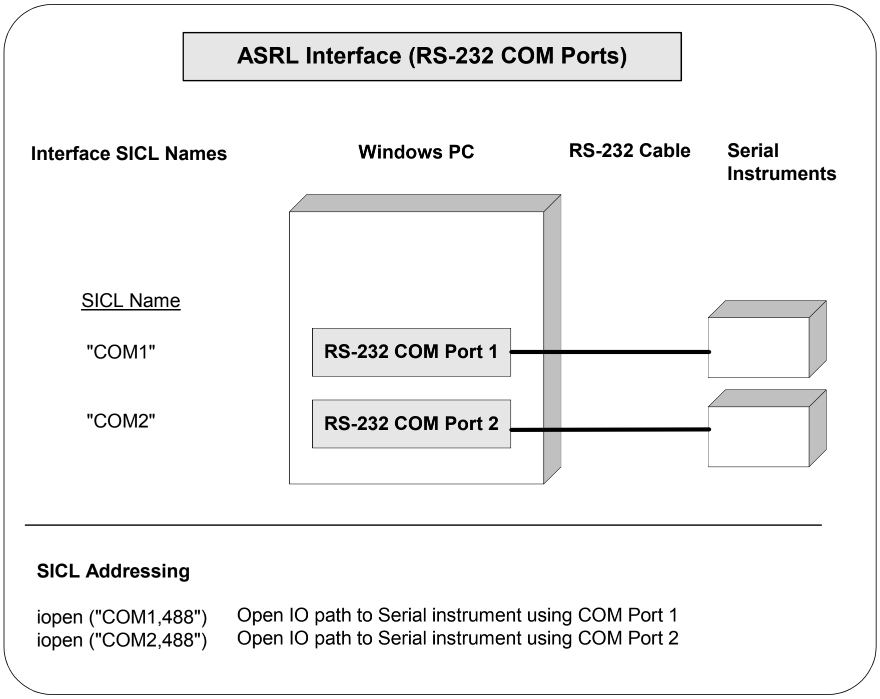
Configuring RS-232 Interfaces

An **IO interface** can be defined as both a hardware interface and as a software interface. The purpose of the IO Config utility is to associate a unique interface name with a hardware interface.

The IO Libraries use 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. IO Config assigns an Interface Name and Logical Unit Number to the interface hardware, as well as other necessary configuration values for an interface when the interface is configured. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on IO Config.

Example: 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. The IO Config utility 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 IO Config, you can now use the SICL **iopen** command to open the IO paths to the GPIB instruments as shown in the figure.



```
iopen ("COM1,488")   Open IO path to Serial instrument using COM Port 1
iopen ("COM2,488")   Open IO path to Serial instrument using COM Port 2
```

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 for Windows uses the RS-232 facilities built into the Windows operating system, controlling RS-232 instruments is easy.

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

Device 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 39 The iserialctrl Functions

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

Table 40 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
I_SERIAL_FLOW_CTRL	Flow control	I_SERIAL_FLOW_NONE I_SERIAL_FLOW_XON I_SERIAL_FLOW_RTS_CTS I_SERIAL_FLOW_DTR_DSR

Table 40 iserialctrl Sets the State for these RS-232 Characteristics

Request	Characteristic	Settings
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 41 iserialstat

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

Table 42 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_*
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

Table 42 iserialstat Captures Status for these RS-232 Characteristics

Request	Characteristic	Value
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 43 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.
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 IO Config utility. To open IO Config, click the Agilent IO Libraries Control **IO** icon on the taskbar and then click **Run IO Config**. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on IO Config.

Some example addresses for RS-232 device sessions:

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

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

- C example:

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

- Visual Basic example:

```
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 44 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)
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.

Example Device Session Programs

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

Example: RS-232 Device Session (C)

This example program takes a measurement from a DVM using a SICL device session. This example program was tested with a 34401A Digital Voltmeter. When you run the program with a Serial connection to the 34401A, be sure that DTR/DSR flow control is set for the Serial port. Otherwise, the program will *appear* not to work.

```
/* ser_dev.c
This example program takes a measurement from a
DVM using a SICL device session.*/

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

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

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

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

main()
  {
  INST dvm;
  double res;

  #if defined(__BORLANDC__) &&
    !defined(__WIN32__
    _InitEasyWin();/*Required for Borland
                     EasyWin*/
  #endif
  /* Log message and terminate on error */
  ionerror (error_handler);

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

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

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

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

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

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

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

  return 0;
  }
```

Example: RS-232 Device Session (Visual Basic)

This example program takes a measurement from a DVM using a SICL device session. This example program was tested with a 34401A Digital Voltmeter. When you run the program with a Serial connection to the 34401A, be sure that DTR/DSR flow control is set for the Serial port. Otherwise, the program will *appear* not to work.

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

Table 45 Sample RS-232 Addresses

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

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

- C example:

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

- Visual Basic example:

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

SICL Functions for RS-232 Interface Sessions

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

Table 46 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 IO Config utility. Any transition on the designated service request line will cause an SRQ handler in your program to be called. (Be sure not to set the SRQ line to CTS or DSR if you are also using that line for hardware flow control.) Service requests are supported for both device sessions and interface sessions. When the designated SRQ line changes state, the RS-232 driver calls all SRQ handlers installed by either device sessions or interface sessions.

Table 46 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**.
iserial-mclctrl	Controls the modem control lines RTS and DTR. If one of these lines is being used for flow control, you cannot set that line with this function.

Table 46 Implementing Some SICL Functions for RS-232

iserial-mclstat	Determines the current state of the modem control lines. If one of these lines is being used for flow control, this function may not give the correct state of that line.

Example Interface Sessions Programs

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

Example: RS-232 Interface Session (C)

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

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

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

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

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

  /* open RS-232 interface session */
  intf = iopen ("COM1");
  itimeout (intf, 10000);

  /* get baud rate, parity, data and stop bits */
  iserialstat (intf, I_SERIAL_BAUD, &baudrate);
  iserialstat (intf, I_SERIAL_PARITY, &parity);
  iserialstat (intf, I_SERIAL_WIDTH, &databits);
  iserialstat (intf, I_SERIAL_STOP, &stopbits);

  /* determine string to display for parity */
  if (parity == I_SERIAL_PAR_NONE) parity_str =
    "NONE";
  else if (parity == I_SERIAL_PAR_ODD)
    parity_str = “ODD”;
  else if (parity == I_SERIAL_PAR_EVEN)
    parity_str = “EVEN”;
  else if (parity == I_SERIAL_PAR_MARK)
    parity_str = “MARK”;
  else  /*parity == I_SERIAL_PAR_SPACE*/
    parity_str = “SPACE”;

  /* set to 9600,NONE,8,1 */
  iserialctrl (intf, I_SERIAL_BAUD, 9600);
  iserialctrl (intf, I_SERIAL_PARITY,
    I_SERIAL_PAR_NONE);
  iserialctrl (intf, I_SERIAL_WIDTH,
    I_SERIAL_CHAR_8);
  iserialctrl (intf, I_SERIAL_STOP,
    I_SERIAL_STOP_1);

  /* Display previous settings */
  printf(“Old settings:  %5ld,%s,%ld,%ld\n”,
    baudrate, parity_str, databits, stopbits);

  /* close port */
  iclose (intf);

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

  return 0;
  }
```

Example: RS-232 Interface Session (Visual Basic)

```
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:
' Start | Programs | Agilent IO Libraries | IO
` Config
' Change this to the SICL Name you have
` defined in IO Config

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

For C/C++: `C:\Program Files\Agilent\IO Libraries\c\samples\`

For Visual Basic: `C:\Program Files\Agilent\IO Libraries\vb\samples\`

The chapter includes:

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

Introduction to LAN Interfaces

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

- LAN Interfaces Overview
- Configuring LAN Client Interfaces
- Configuring LAN Server Interfaces

LAN Interfaces Overview

A LAN extends control of instrumentation beyond the limits of typical instrument interfaces. LAN is *only* supported with 32-bit SICL on Windows 98SE, Windows Me, Windows 2000, Windows XP, and Windows NT 4.0. LAN is *only* supported with 32-bit Visual Basic version 4.0 and above. Also, the GPIO interface is ***not*** supported with SICL over LAN.

The LAN software provided with SICL allows instrumentation control over a LAN. By using standard LAN connections, instrument control can be driven from a computer that does not have a special interface for instrument control. To start or stop the LAN server, see the *Agilent IO Libraries Installation and Configuration Guide for Windows.*

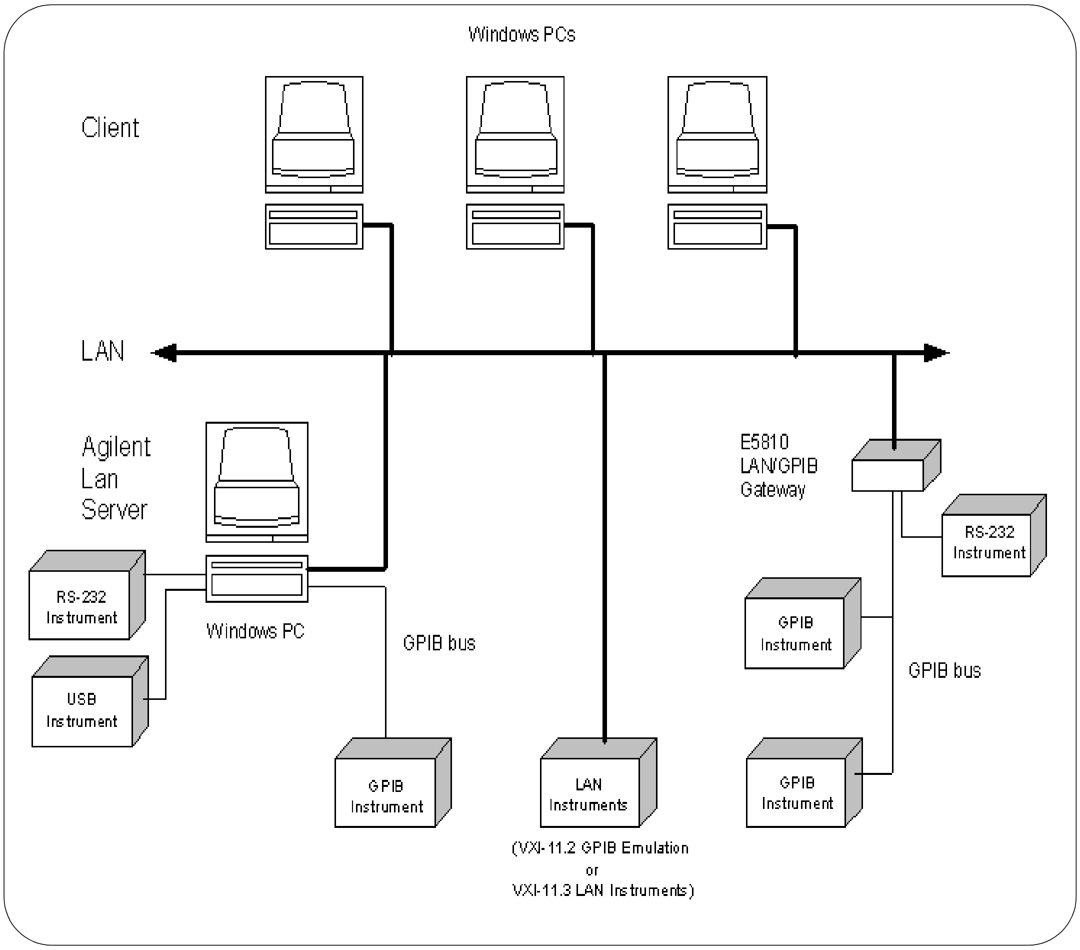
LAN Client/Server Model

The LAN software provided with SICL uses the **client/server** model of computing. **Client/server computing** refers to a model where an application, the **client**, does not perform all the necessary tasks of the application itself. Instead, the client makes requests of another computing device, the **server**, for certain services. Examples that you may have in your workplace include shared file servers, print servers, or database servers.

The use of LAN for instrument control also provides other advantages associated with client/server computing, such as resource sharing by multiple applications/people within an organization, or distributed control, where the computer running the application controlling the devices need not be in the same room (or even the same building) as the devices.

LAN Hardware Architecture

As shown in the following figure, a LAN client computer system makes SICL requests over the network to a LAN server, a Windows PC, or an E5810 LAN/GPIB Gateway).

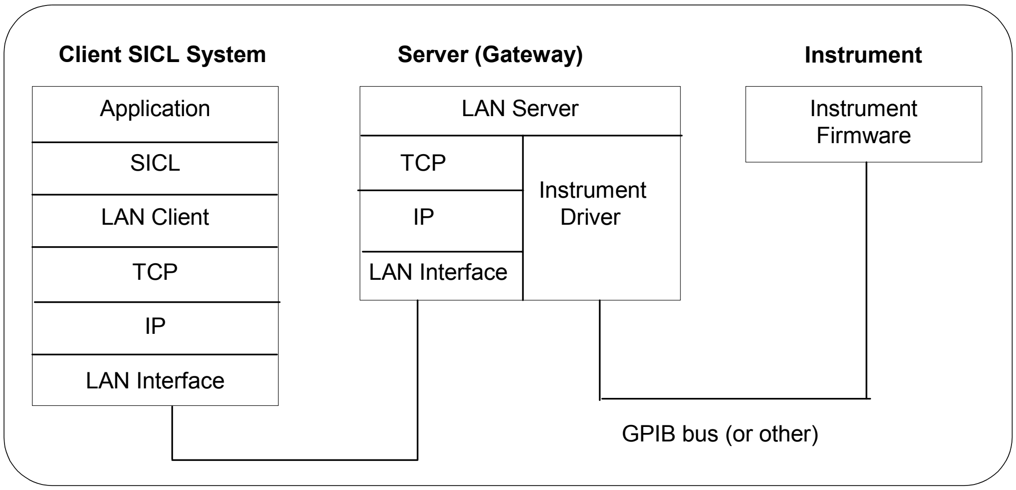


The LAN server is connected to the instrumentation or devices that must be controlled. Once the LAN server has completed the requested operation on the instrument or device, the LAN server sends a reply to the LAN client. This reply contains any requested data and status information that indicates whether the operation was successful.

The LAN server acts as a **gateway** between the LAN that the client system supports, and the instrument-specific interface that the device supports. Due to the LAN server's gateway functionality, we refer to devices or interfaces that are accessed via one of these LAN-to-instrument_interface gateways as being a *LAN-gatewayed* device or a *LAN-gatewayed* interface.

LAN Software Architecture

As shown in the following figure, the client system contains the LAN client software and the LAN software (TCP/IP) needed to access the server (gateway). The gateway contains the LAN server software, LAN (TCP/IP) software, and the instrument driver software needed to communicate with the client and to control the instruments or devices connected to the gateway.



LAN Networking Protocols

The LAN software is built on top of standard LAN networking protocols. There are two LAN networking protocols provided with the Agilent IO Libraries software. You can use one or both of these protocols when configuring your systems (via Agilent IO Libraries configuration) to use SICL over LAN.

- **SICL-LAN Protocol** is a networking protocol developed by Agilent Technologies. This LAN networking protocol is the default choice in the Agilent IO Libraries configuration when configuring the LAN client. The SICL LAN protocol supports SICL operations over LAN/GPIB Gateways (e.g., E5810) and PCs running the Agilent LAN Server. It supports GPIB, RS-232, and USB interfaces.
- **VXI-11 (TCP/IP Instrument Protocol)** is a networking protocol developed by the VXIbus Consortium based on the SICL LAN Protocol that permits interoperability of LAN software from different vendors who meet the VXIbus Consortium standards.

When using either of these networking protocols, the LAN software uses the TCP/IP protocol suite to pass messages between the LAN client and the LAN server. The server accepts device I/O requests over the network from the client and then proceeds to execute those I/O requests on a local interface (GPIB, etc.).

By default, the LAN client supports both protocols by automatically detecting the protocol the server is using. When a SICL **iopen** call is performed, the LAN client driver first tries to connect using the SICL-LAN protocol. If that fails, the driver will try to connect using the VXI-11 protocol.

If you want to control the protocol used, you can configure more than one LAN client interface and set each interface to a different protocol. The protocol used will then depend on the interface you are connecting through. Thus, you can have more than one SICL-LAN and/or VXI-11 interface configured for your system.

In SICL, the programmer can override the configuration settings 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 LAN client 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.'

When you have configured VISA LAN client interfaces, you can then use the interface name specified during configuration in an **iopen** call of your program. However, the LAN server does *not* support simultaneous connections from LAN clients using the SICL-LAN Protocol and from LAN clients using VXI-11 (TCP/IP Instrument Protocol).

There are four LAN servers that can be used with SICL: the E2050 LAN/GPIB Gateway, E5810 LAN/GPIB Gateway, an HP Series 700 computer running HP-UX, or a PC running Windows 98SE/Me/2000/XP/NT. To use this capability, the LAN server on the PC must have a local GPIB, RS-232, and/or USB interface configured for I/O.

LAN Clients and Threads

You can use multi-threaded designs (where SICL calls are made from multiple threads) in WIN32 SICL applications over LAN. However, only one thread is permitted to access the LAN driver at a time. This sequential handling of individual threads by the

LAN driver prevents multiple threads from colliding or overwriting one another. Requests are handled sequentially even if they are intended for different LAN servers.

Use multiple processes to process concurrent threads simultaneously with SICL over LAN. see Chapter 3, "Programming with SICL" for more information on using threads in SICL applications. Also, see "Using Locks and Threads Over LAN" in this chapter for information on using locks in multi-threaded applications.

LAN Servers

SICL includes software required to allow a Windows 98SE/Me/2000/XP/NT PC to act as a LAN-to-instrument_interface gateway. To use this capability, the PC must have a local interface configured for I/O. The supported interfaces are GPIB, RS-232, and USB with the SICL-LAN Protocol, and GPIB with the VXI-11 Protocol. The LAN server does *not* support VXI operations with either protocol.

Timing of operations performed remotely over a network will be different from timing of operations performed locally. The extent of the timing difference will, in part, depend on the bandwidth of, and the traffic on, the network being used.

SICL LAN Configuration and Performance

As with other client/server applications on a LAN, when deploying an application that uses SICL over LAN, consideration must be given to the performance and configuration of the network to which the client and server will be attached. If the network to be used is not a dedicated LAN or otherwise isolated via a bridge or other network device, current use of the LAN must be considered.

Depending on the amount of data to be transferred over the LAN via the SICL application, performance problems could be experienced by the SICL application or other network users if sufficient bandwidth is not available. This is not unique to SICL over LAN, but is a general design consideration when deploying any client/server application.

If you have questions concerning the ability of your network to handle SICL traffic, consult with your network administrator or network equipment providers. If you are connecting to a VXI-11 device, you can configure a VXI-11 interface (rather than AUTO) in IO Config and connect through it to achieve slightly better **iopen** performance.

If an attempt is made to **iopen** a session on a LAN host that is turned off or not connected to the network, it may take up to several minutes for the **iopen** to return a failure. This delay is not affected by any of the timeout parameters that can be configured in SICL or by the IO Config utility.

SICL LAN Functions

This table summarizes the SICL functions for the LAN interface.

Table 47 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.

Configuring LAN Client Interfaces

An **IO interface** can be defined as both a hardware interface and as a software interface. The purpose of the IO Config utility is to associate a unique interface name with a hardware interface.

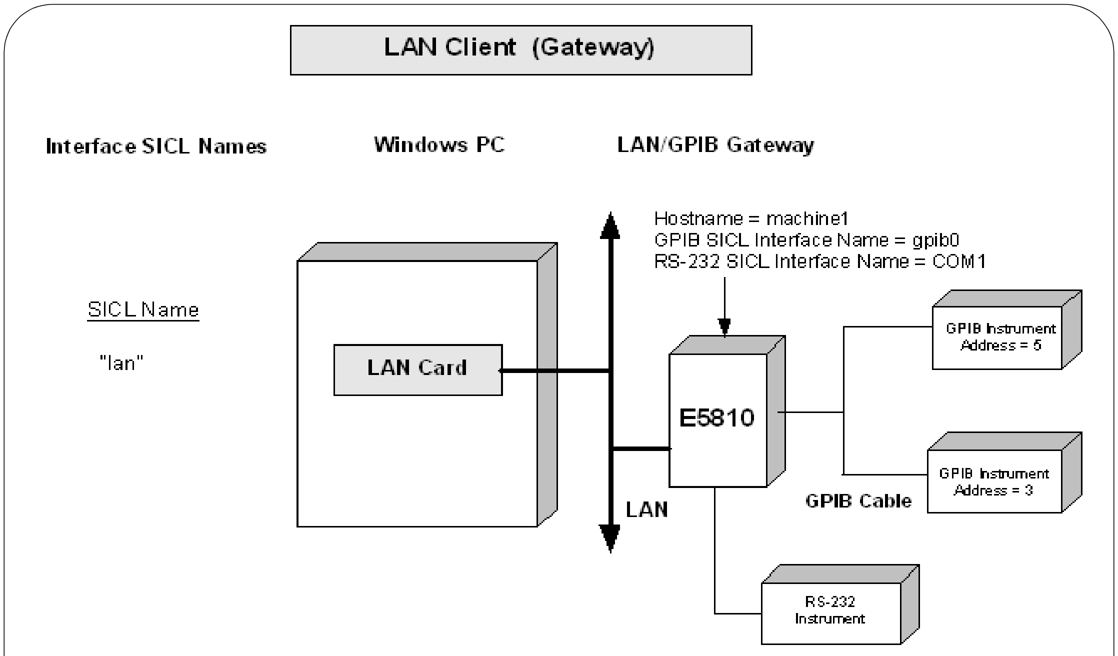
Using IO Config

The IO Libraries use an **Interface Name** or **Logical Unit Number** to identify an interface. This information is passed in the parameter string of the **viOpen** function call in a VISA program or in the **iopen** function call in a SICL program. IO Config assigns an Interface Name and Logical Unit Number to the interface hardware, as well as other necessary configuration values for an interface when the interface is configured. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for details on IO Config.

Example: Configuring LAN Client (Gateway) Interface

The LAN client interface system in the following figure consists of a Windows PC with a LAN card, an E5810 LAN/GPIB gateway, two GPIB instruments, and an RS-232 instrument. For this system, the IO Config utility has been used to assign the LAN card a SICL name of **lan**.

With this name assigned to the interface, SICL addressing is as shown in the figure. Since a unique name has been assigned by IO Config, you can now use the **iopen** command to open the I/O paths to the GPIB instruments as shown in the figure.



SICL Addressing (Using LAN Client)

SICL-LAN or VX2-11 Protocol

iopen ("lan [machine1]:gpib0,5")	Open IO path to GPIB instrument at address 5
iopen ("lan [machine1]:gpib0,3")	Open IO path to GPIB instrument at address 3

SICL-LAN Protocol only

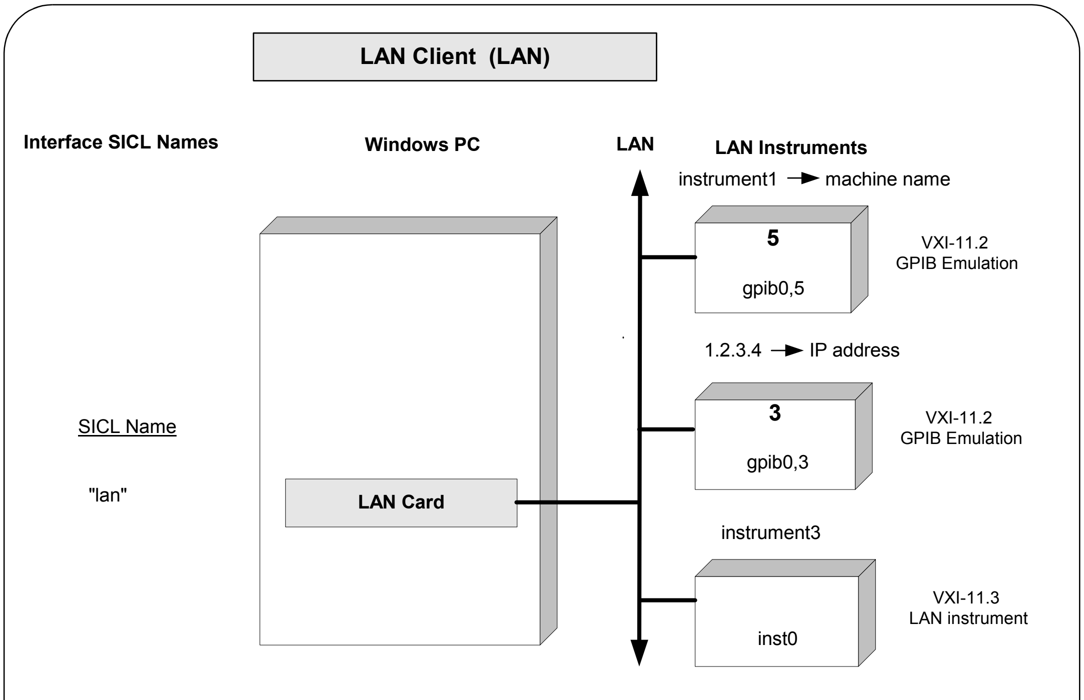
iopen ("lan [machine1]:COM1,488")	Open a path to an RS-232 instrument

Example: Configuring LAN Client (LAN) Interface

The LAN client interface system in the following figure consists of a Windows PC with a LAN card and three LAN instruments. Instrument1 and instrument2 are VXI-11.2 (GPIB Emulation) instruments and instrument3 is a VXI-11.3 LAN instrument.

For this system, the IO Config utility has been used to assign the LAN card a SICL name of **lan**. For the addressing examples, instrument1 has been addressed by its machine name, instrument 2 has been addressed by its IP address, and instrument3 by its LAN name (**inst0**).

Since unique names have been assigned by IO Config, you can now use the **iopen** command to open the I/O paths to the GPIB instruments as shown in the figure.



SICL Addressing (Using LAN Client)

iopen ("lan [instrument1]:gpib0,5")	Open IO path to LAN instrument at address 5
iopen ("lan [1.2.3.4]:gpib0,3")	Open IO path to LAN instrument at address 3
iopen ("lan [instrument3]:inst0")	Open IO path to LAN instrument3

Configuring LAN Server Interfaces

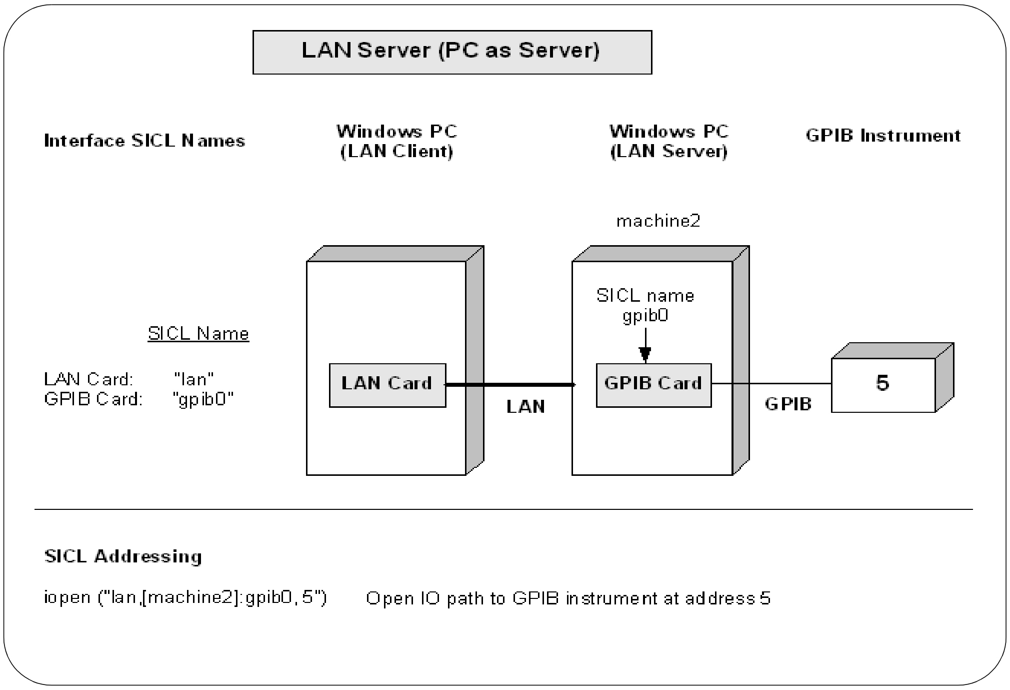
An **IO interface** can be defined as both a hardware interface and as a software interface. The purpose of the IO Config utility is to associate a unique interface name with a hardware interface.

Using IO Config

The IO Libraries use an **Interface Name** or **Logical Unit Number** to identify an interface. This information is passed in the parameter string of the **viOpen** function call in a VISA program or in the **iopen** function call in a SICL program. IO Config assigns an Interface Name and Logical Unit Number to the interface hardware, as well as other necessary configuration values for an interface when the interface is configured. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for details on IO Config.

Example: Configuring LAN Server Interface

The LAN Server interface system in the following figure consists of a Windows PC acting as a LAN client, a second PC acting as a LAN server, and a GPIB instrument. The IO Config utility has been used to assign the LAN card a SICL name of **lan**. Also, the GPIB card in the LAN server PC has been assigned SICL name of **gpib0**. The LAN server PC has been assigned a name of **machine2**. Since unique names have been assigned by IO Config, you can now use the **iopen** command to open the IO paths to the GPIB instruments as shown in the figure.


LAN Server (PC as Server)
Interface SICL Names
Windows PC (LAN Client)
Windows PC (LAN Server)
GPIB Instrument
machine2
SICL Name
LAN Card: "lan"
GPIB Card: "gpib0"
LAN Card
SICL name gpib0
GPIB Card
LAN
GPIB
5
SICL Addressing
iopen ("lan,[machine2]:gpib0,5")
Open IO path to GPIB instrument at address 5

Using LAN-gatewayed Sessions

This section provides guidelines to using LAN-gatewayed Sessions, including:

- Addressing Guidelines
- SICL Function Support
- Example Programs

Addressing Guidelines

Communicating with a device over a LAN via a LAN-to-instrument_interface gateway preserves the functionality of the gatewayed-interface, with a few exceptions. Thus, most operations over an interface (such as GPIB connected directly to your controller), can also be performed over a remote interface via the LAN gateway.

The only portions of your application that must be changed are the addresses passed to the **iopen** calls (unless those addresses are stored in a configuration file, in which case no changes to the application itself are required). The address used for a local interface must have a LAN prefix added so the SICL software knows to direct the request to a LAN server on the network.

Creating a LAN-gatewayed Session

To create a LAN-gatewayed session, specify the LAN's interface logical unit or interface name, the IP address or hostname of the server machine, and the address of the remote interface or device in the *addr* parameter of the **iopen** function. The interface logical unit and interface name are defined by running the IO Config utility.

To open the the IO Config utility, click the Agilent IO Libraries Control IO icon on the taskbar and then click **Run IO Config**. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on running IO Config.

Example: LAN-gatewayed Addressing

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

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

Table 48 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 IO Config 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 IO Config will be used. This is the same as not specifying a protocol.

Table 48 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 49 Relationships Between SICL Functions

Address	Session Type	Interface Type	Gateway Type
lan	I_SESS_INTF	I_INTF_LAN	I_INTF_NONE
lan[instserv]:gpib0	I_SESS_INTF	I_INTF_GPIB	I_INTF_LAN
lan[instserv]:gpib0,7	I_SESS_DEV	I_INTF_GPIB	I_INTF_LAN
gpib0	I_SESS_INTF	I_INTF_GPIB	I_INTF_NONE
gpib0,7	I_SESS_DEV	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 was local, with the following exceptions or qualifications.

Table 50 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 names at the remote server may be configurable, this is not guaranteed. Correct behavior of **iremote** and **iclear** depend on the correct address strings being used. When **iremote** is executed over the VXI-11 protocol, **iremote** also 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 timeout over LAN, even those functions that cannot timeout over local interfaces. (See “Using Timeouts with LAN” in this chapter for more details.) These functions all cause a request to be sent to the server for execution.

All GPIB specific functions All RS-232/serial specific functions iabort, iclear, iclose, iflush, ifread, ifwrite, igetintfsess, ilocal, ilock, ionintr, ionsrq, iopen, iprintf, ipromptf, iread, ireadstb, iremote, iscanf, isetbuf, isetintr, isetstb, isetubuf, itrigger, iunlock, iversion, iwrite, ixtrig

These SICL functions perform as follows with LAN-gatewayed sessions.

Table 51 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.

Example Programs

Two example programs for LAN-gatewayed sessions follow, one for C Language and one for Visual Basic.

Example: LAN-gatewayed Session (C) This example program opens a GPIB device session via a LAN-to-GPIB gateway. This example is the same as the example in *Chapter 4 - Using SICL with GPIB*, except the addresses passed to the **iopen** calls are modified. The addresses in this example assume a machine with hostname **instserv** is acting as a LAN-to-GPIB gateway.

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


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

main(){

  INST dvm;
  INST sw;
  double res;
  int i;

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

  /* Open the multimeter and switch sessions */
  dvm = iopen ("lan[instserv]: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);
}
```

Example: LAN-gatewayed Session (Visual Basic)

This example 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” in this chapter.)

Addressing LAN Interface Sessions

To create a LAN interface session, specify the interface logical unit or interface name in the *addr* parameter of the **iopen** function. The interface logical unit and interface name are defined by running the IO Config utility.

To open the the IO Config utility, click the Agilent IO Libraries Control **IO** icon on the taskbar and then click **Run IO Config**. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on running IO Config. Some examples of LAN interface addresses follow.

Table 52 LAN Interface Address 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 and return I_ERR_NOTSUPP.

All GPIB specific functions
All serial specific functions
All formatted I/O routines
iwrite, iread, ilock, iunlock, isetintr, itrigger, ixtrig,
ireadstb, isetstb, imapinfo, ilocal, iremote

These SICL functions perform as follows with LAN interface sessions.

Table 53 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 interfaces that are being accessed via a LAN-to-instrument_interface gateway.

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*.

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

Recommended Usage

You can avoid each scenario by always using the same strings to identify the same device or interface in multi-threaded applications. You can also use the **igetintfsess** function if other sessions use the logical unit to specify the interface instead of the interface's symbolic name.

If any thread uses **ilock** and **iunlock** to synchronize access to a particular device or interface, all threads accessing that same device or interface using a different session must also use **ilock** and **iunlock**. WIN32 synchronization techniques may also be used to ensure that a thread does not attempt I/O (**iread/iwrite**, etc.) to a device already locked via a different session from a different thread within the same process.

If a session has an interface locked, and if a different thread using its own session attempts to lock a device on that interface, the device lock will be held off either until the interface is unlocked by the other thread, or until a timeout occurs on the device lock. This is different from how **ilock** works on other interfaces (where a lock on a device when the device's interface is already locked will not hold off the **ilock** operation, but rather will hold off any subsequent I/O to the device).

Using Timeouts with LAN

The client/server architecture of the LAN software requires use of two timeout values: one for the client and one for the server. The server's timeout value is the SICL timeout value specified with the **itimeout** function. The client's timeout value is the LAN timeout value, which may be specified with the **ilantimeout** function.

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 the IO Config utility.

Once **ilantimeout** is called by the application, 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

The IO Config utility specifies two timeout-related configuration values for the LAN software. These values are used by the software to calculate timeout values if the application has not previously called **ilantimeout**.

Table 54 LAN Software Timeout Values

Server Timeout	Timeout value passed to the server when an application either uses the SICL default timeout value of infinity or sets the SICL timeout to infinity (0). Value specifies the number of seconds the server will wait for the operation to complete before returning I_ERR_TIMEOUT. A value of 0 in this field will cause the server to be sent a value of infinity if the client application also uses the SICL default timeout value of infinity or sets the SICL timeout to infinity (0).
Client Timeout Delta	Value added to the SICL timeout value (server's timeout value) to determine the LAN timeout value (client's timeout value). Value specifies the number of seconds.

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 Timeout Delta is greater than the current LAN timeout. In that case the LAN timeout will be set to the SICL timeout plus the Client Timeout Delta.
- The calculated LAN timeout only increases as necessary to meet the needs of the application, but never decreases. This avoids the overhead of readjusting the LAN timeout every time the application changes the SICL timeout.

- The first **iopen** call used to set up the server connection uses the Client Timeout Delta specified via the IO Config utility for portions of the **iopen** operation. The timeout value for TCP connection establishment is not affected by the Client Timeout Delta.

To change the defaults:

1 Exit any LAN applications for SICL to be reconfigured.

2 Run the IO Config utility. (Click the Agilent IO Libraries Control and then click **Run IO Config**.)

3 Change the Server Timeout and/or Client Timeout Delta value(s).

4 Restart the LAN applications for SICL.

Timeouts in Multi-threaded Applications

To manually set the client-side timeout in an application using multiple threads, be aware that **ilantimeout** may itself timeout due to contention for the LAN subsystem where multiple threads in an application are simultaneously using SICL over LAN.

Thus, if multiple threads are using SICL over LAN at the same time and LAN timeouts are expected by the application, it is recommended that **ilantimeout** be called when no other LAN I/O is occurring, such as immediately after session creation (**iopen**).

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

Timeout Configurations to Be Avoided

The LAN timeout used by the client should always be set greater than the SICL timeout used by the server. This avoids the situation where the client times out while the server continues to attempt the request, potentially holding off subsequent operations from the same client. This also avoids having the server send unwanted replies to the client.

The SICL timeout used by the server should generally be less than infinity. Having the LAN server wait less than forever allows the LAN server to detect network problems or clients that have ceased operation abruptly, and subsequently release resources associated with those clients, such as locks.

Using the smallest possible value for your application will maximize the server's responsiveness to dropped connections, including the client application being terminated abnormally. Setting a value less than infinity is done by setting the Server Timeout configuration value via the IO Config utility.

Even if your application uses the SICL default of infinity, or if **itimeout** is used to set the timeout to infinity, by setting the Server Timeout value to some reasonable number of seconds, the server will be allowed to timeout, detect network trouble, and release resources.

Application Terminations and Timeouts

If an application is stopped in the middle of a SICL operation performed at the LAN server, the server continues to try the operation until the server's timeout is reached. By default, the LAN server associated with an application that is using a timeout of infinity and is stopped may not discover that the client is no longer running for 2 minutes. For a server other than the LAN server on HP-UX, Windows 98SE/Me/2000/XP/NT, or the E5810, check that server's documentation for its default behavior.

If **itimeout** is used by the application to set a long timeout value, or if both the LAN client and LAN server are configured to use infinity or a long timeout value, the server may appear "hung." If this situation occurs, the LAN client (via the Client Timeout Delta value set with the IO Config utility) or the LAN server (via its Server Timeout value) may be configured to use a shorter timeout value.

If long timeouts must be used, the server may be reset. A LAN server may be reset by logging into the server system and stopping the LAN server that is running. The latter procedure will affect all clients connected to the server. See *"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

USBTMC/USBTMC-USB488 instruments are detected and automatically configured by the Agilent IO Libraries when they are plugged into the computer. The *Agilent IO Libraries Installation and Configuration Guide for Windows* 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 the Agilent IO Libraries.

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 55 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, the Agilent IO Libraries automatically configures a USB interface with the name **usb0** if one does not already exist. (See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for more details.) A dialog box is also displayed, showing an “alias” name (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 number 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 56 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 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 IO Config and selecting the USB Interface in the **Configured Interfaces** list box, and then clicking **Edit**.

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

Although the case of an alias name 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 name, 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 file for specific information on these functions.)

Table 57 Operations Supported on All USBTMC Devices

Function Name	Action
iusbctrl	Used to set parameters affecting the USB device.
iusbgetcapabilities	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.

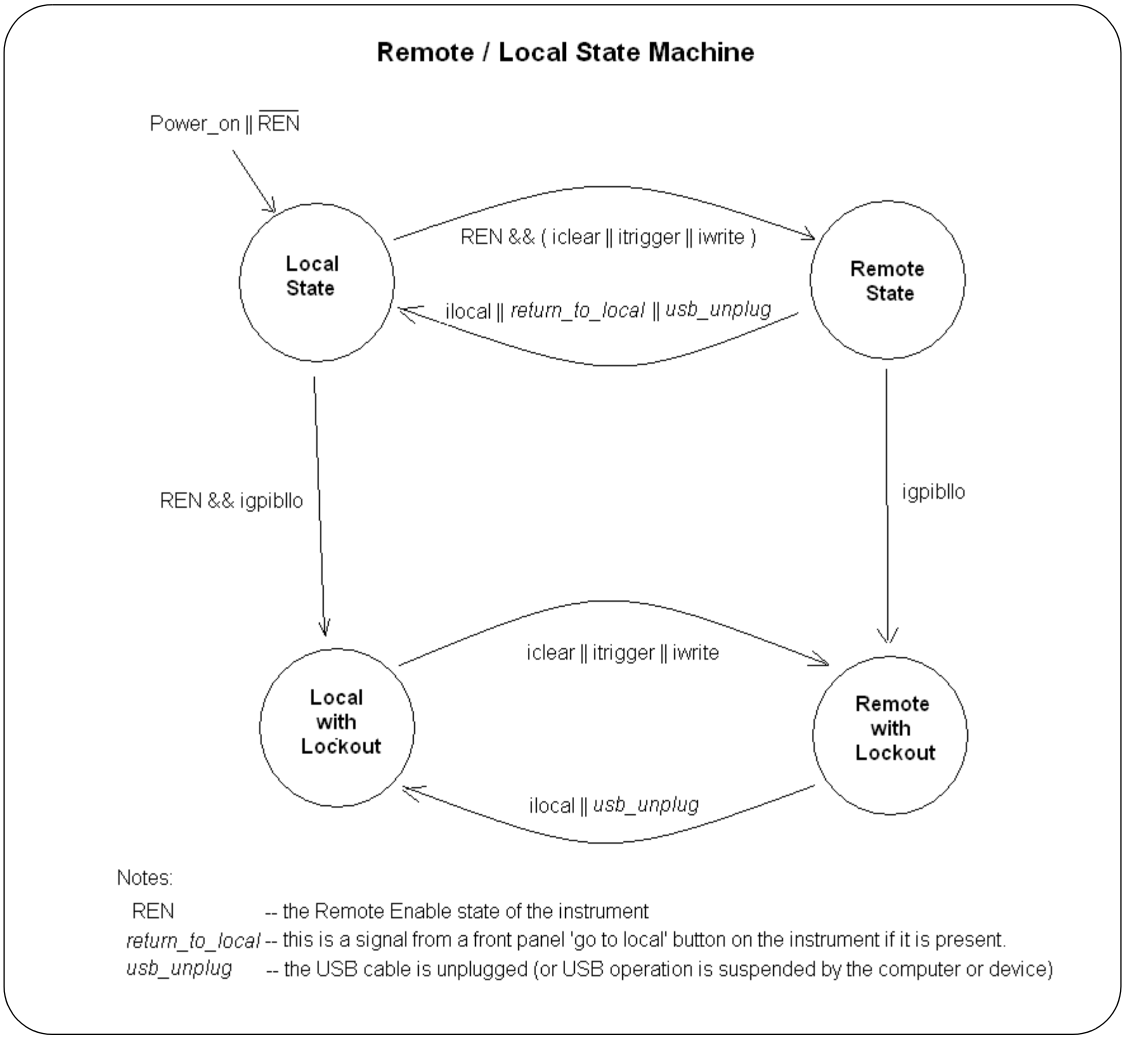
Interrupts are not supported on USBTMC or on USBTMC-USB488 devices.

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 file 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.)

SRQ's (*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.


Remote / Local State Machine
Power_on || $\overline{\text{REN}}$
Local State
Remote State
REN && (iclear || itrigger || iwrite)
ilocal || *return_to_local* || *usb_unplug*
REN && igpibllo
igpibllo
Local with Lockout
Remote with Lockout
iclear || itrigger || iwrite
ilocal || *usb_unplug*

Notes:

REN	-- the Remote Enable state of the instrument
return_to_local	-- this is a signal from a front panel 'go to local' button on the instrument if it is present.
usb_unplug	-- the USB cable is unplugged (or USB operation is suspended by the computer or device)

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

Table 58 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 system interaction in Windows 98SE, Windows Me, Windows 2000, Windows XP, and Windows NT. This information can be used as a reference for removing SICL from a system, if necessary. The appendix contents are:

- File System Information
- Porting to Visual Basic
- RS-232 Cabling Information

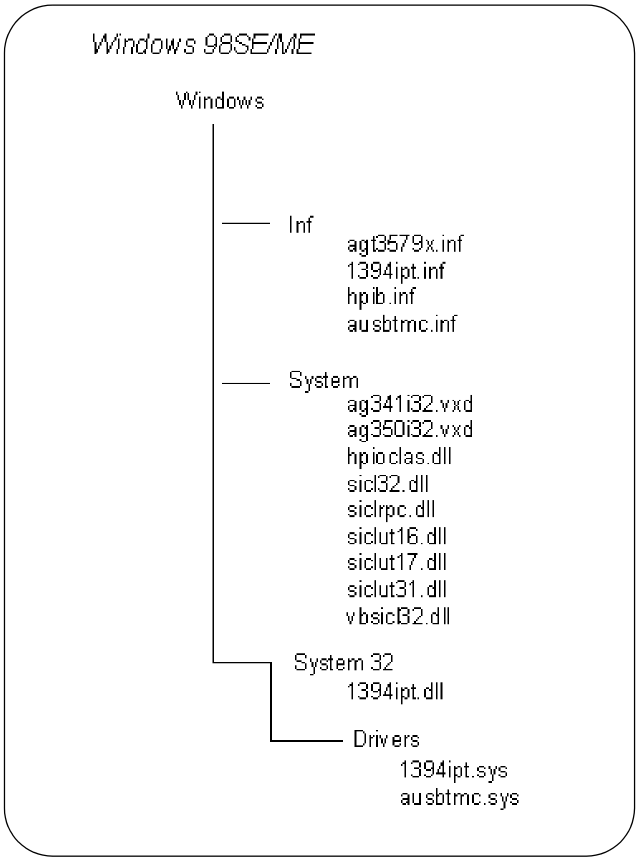
File System Information

This section describes SICL file system information for Windows 98SE, Windows Me, Windows 2000, Windows XP, and Windows NT.

Windows 98/Windows Me

File Location

All SICL files are installed in the base directory specified by the person who installs SICL, with the exception of several common files that Windows must be able to locate. On Windows 98SE and Windows Me, the following files are copied to the **Windows** subdirectory.



The Registry

SICL places the following key in the Windows 98SE or Windows Me registry under HKEY_LOCAL_MACHINE:

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

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

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

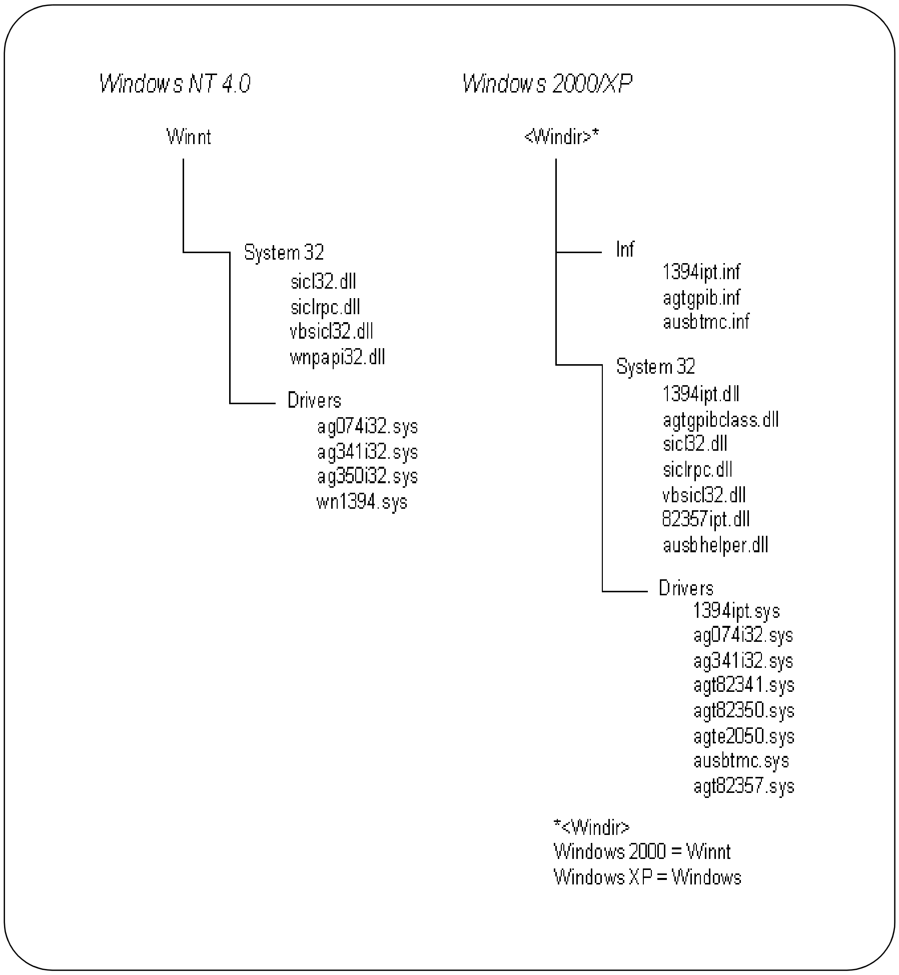
SICL Configuration Information

SICL configuration information is stored in the Windows 98SE or Windows Me registry under the `Software\Agilent\IO Libraries\CurrentVersion` branch under HKEY_LOCAL_MACHINE.

Windows NT/Windows 2000/Windows XP

File Location

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



The Registry

SICL places the following keys in the Windows NT registry under HKEY_LOCAL_MACHINE:

- `Software\Agilent\IO Libraries\CurrentVersion`
- `System\CurrentControlSet\Control\GroupOrderList`
- `System\CurrentControlSet\Control\ServiceOrderList`
- `System\CurrentControlSet\Services\hp341i32`
- `System\CurrentControlSet\Services\EventLog\Application\SICL Log`
- `System\CurrentControlSet\Services\EventLog\System\hp341i32`

SICL Configuration Information

SICL configuration information is stored in the Windows NT or Windows 2000 registry under the `Software\Agilent\IO Libraries\CurrentVersion` branch under HKEY_LOCAL_MACHINE.

Porting to Visual Basic

This section shows how to program SICL applications in Visual Basic version 4.0 or later. For SICL applications written in an earlier Visual Basic version than version 4.0 (for example, version 3.0), you can port your SICL applications to Visual Basic version 4.0 or later.

Porting SICL applications to Visual Basic 4.0 or later is a matter of adding the *SICL32.BAS* declaration file (rather than the *SICL.BAS* file) to each project that calls SICL for Visual Basic 4.0 or later programs. There may also be changes in functions when passing null pointers for strings to SICL functions. For example, in Visual Basic version 3.0, the preceding **ByVal** keyword was used as follows:

```
ivprintf(id, mystring, ByVal 0&)
```

In Visual Basic version 4.0 or later, you only need to pass the *0&* null pointer because version 4.0 or later knows this is by reference:

```
ivprintf(id, mystring, 0&)
```

Once you have added the *SICL32.BAS* declaration file to each project and removed `ByVal` keywords preceding null pointers for strings, your SICL applications will run correctly with Visual Basic 4.0 or later.

RS-232 Cabling Information

This section lists several general purpose RS-232 cables and adapters. Consult your instrument's operating manual for information on the status lines used for handshaking.

Cable/Adapter Part Numbers

In the following table, recommended cables and adapters are shown in **boldface** type. Other cables are listed since they may work better than the recommended cable/adapter in some applications. In the table, “a” and “b” are defined as:

- [a] One of four adapters in the *34399A RS-232 Adapter Kit*. Kit includes four adapters to go from DB9 Female Cable (34398A) to PC/Printer DB25 Male or Female, or to modem DB9 Female or DB25 Female.
- [b] Part of *34398A RS-232 Cable Kit*. Kit comes with RS-232, 9-pin Female to 9-pin Female Null modem/printer cable and one adapter 9-pin Male to 25-pin Female (part number 5181-6641). The adapter is also located in the *34399A RS-232 Adapter Kit*.

Table 59 RS-232 Interface Types and Matching Part Numbers

Instrument Connector	Computer/Printer Connector	Cable Part Number	Adapter Part Number	Length
9-Pin Male	25-Pin Male	24542H 24542U F1047-80002 [b]	none 5181-6641 [a] 5181-6641 [a]	3m (9ft 10in) 3m (9ft 10in) 2.5m (8ft 2.5in)
9-Pin Male	25-Pin Female	24542G 24542U F1047-80002 [b]	none 5181-6640 [a] 5181-6640 [a]	3m (9ft 10in) 3m (9ft 10in) 2.5m (8ft 2.5in)
9-Pin Male	9-Pin Male	24542U 24542H & 24542M F1047-80002 [b]	none none none	3m (9ft 10in) 6m (19ft 10in) 2.5m (8ft 2.5in)
9-Pin Male	25-Pin Female	24542M 24542U F1047-80002 [b]	none 5181-6642 [a] 5181-6642 [a]	3m (9ft 10in) 3m (9ft 10in) 2.5m (8ft 2.5in)
9-Pin Male	9-Pin Female	24542U F1047-80002 [b]	5181-6639 [a] 5181-6639 [a]	3m (9ft 10in) 2.5m (8ft 2.5in)
25-Pin Female	25-Pin Female	24542G	5181-6642 [a]	3m (9ft 10in)
25-Pin Female	9-Pin Female	24542G 24542M	5181-6639 [a] none	3m (9ft 10in) 3m (9ft 10in)
25-Pin Female	25-Pin Male	17255D C2913A 24542G	none none 5181-6641 [a]	1.2m (3ft 11in) 1.2m (3ft 11in) 3m (9ft 10in)

Table 59 RS-232 Interface Types and Matching Part Numbers

Instrument Connector	Computer/Printer Connector	Cable Part Number	Adapter Part Number	Length
25-Pin Female	25-Pin Female	13242G	none	5m (16ft 8in)
		17255M	none	1.5m (4ft 11in)
		C2914A	none	1.2m (3ft 11in)
		24542G	5181-6640 [a]	3m (9ft 10in)
25-Pin Female	9-Pin Male	24542G	none	3m (9ft 10in)
		24542U	5181-6640 [a]	3m (9ft 10in)
		F1047-80002 [b]	5181-6640 [a]	2.5m (8ft 2.5in)

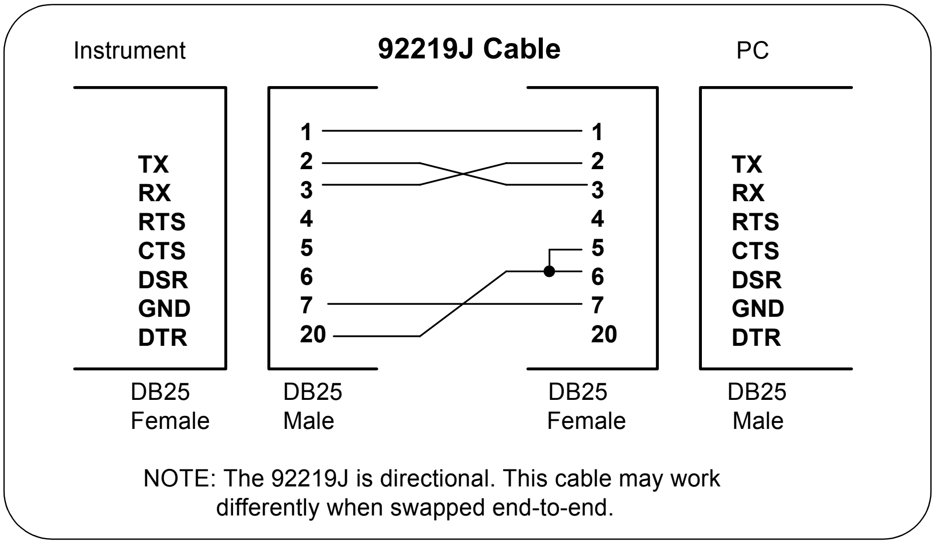


Figure 1 Cable/Adapter Pinouts

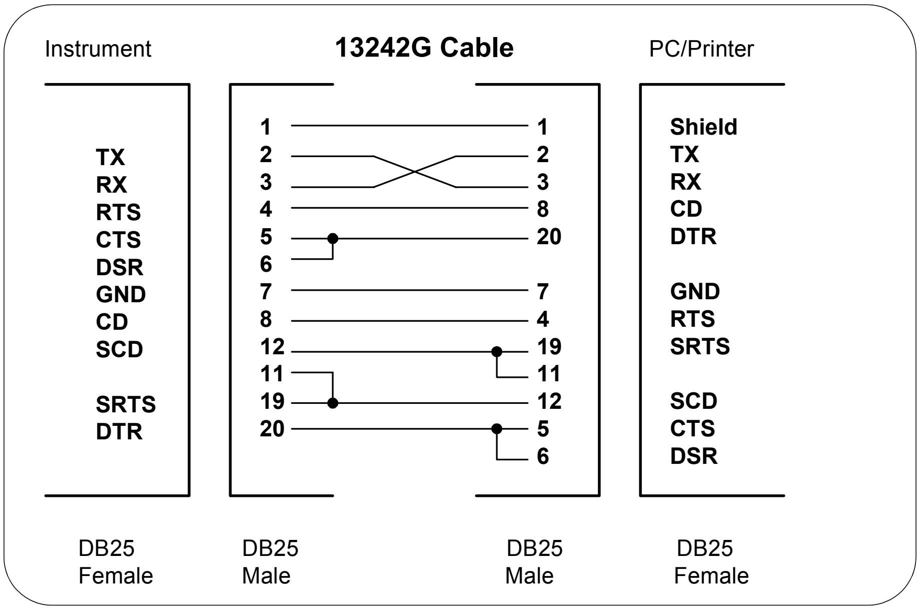


Figure 2 13242G Cable

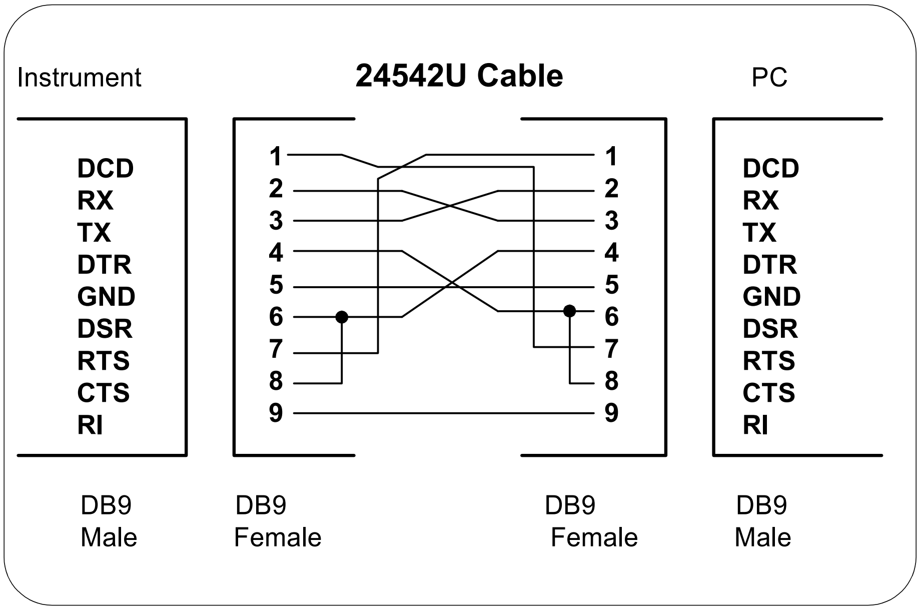


Figure 3 24542U Cable

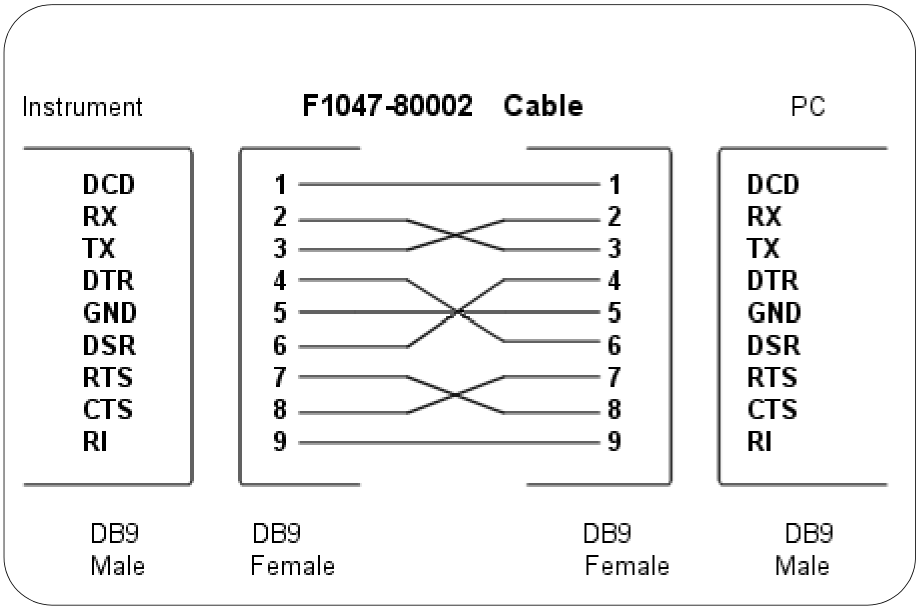


Figure 4 F1047-80002 Cable

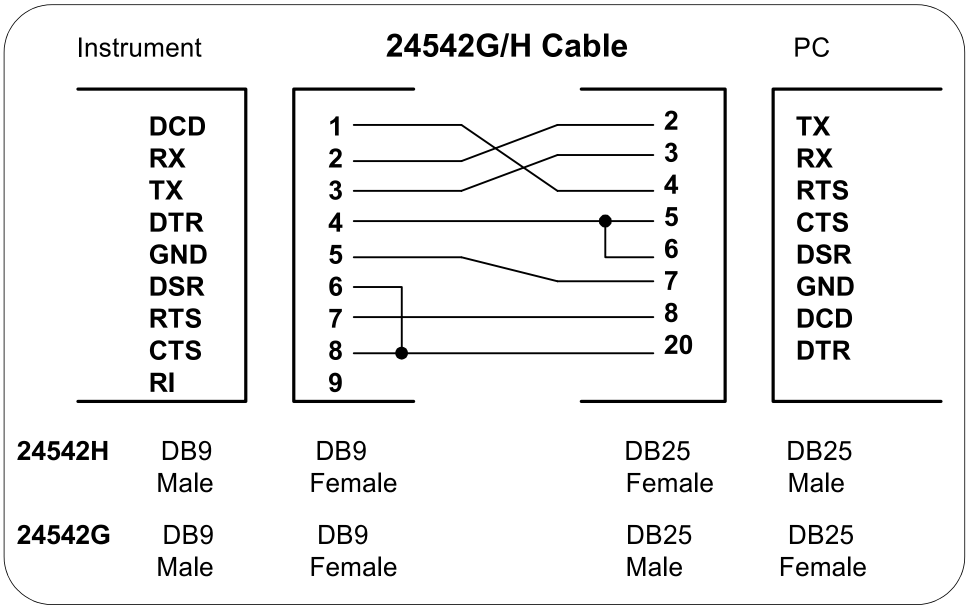


Figure 5 24542G/H Cable

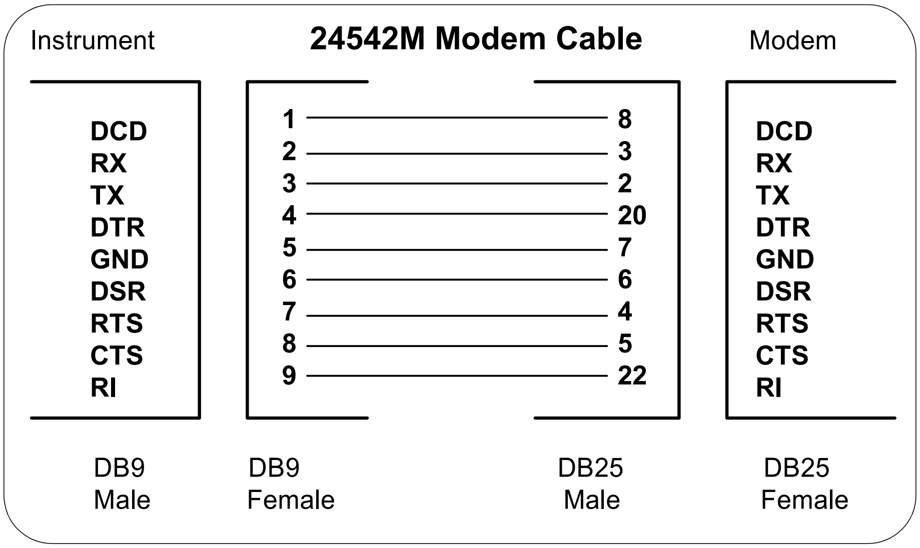


Figure 6 24542 Modem Cable

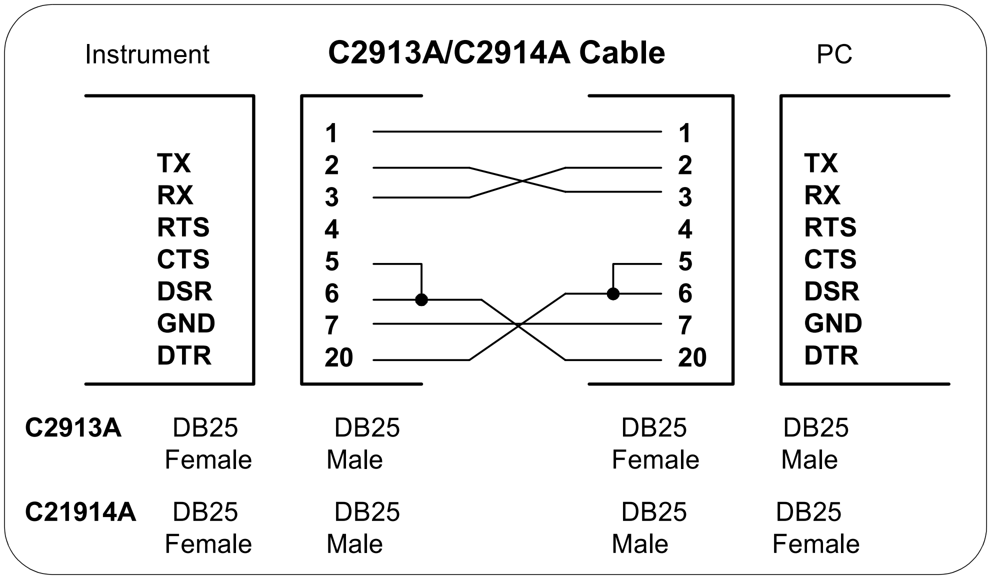


Figure 7 C2913A/2914A Cable

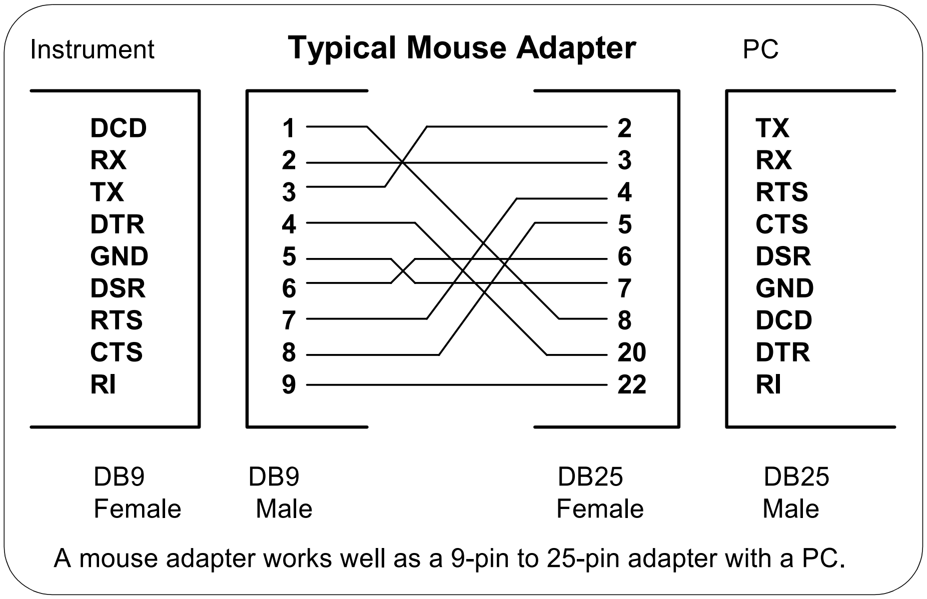


Figure 8 Typical Mouse Adapter

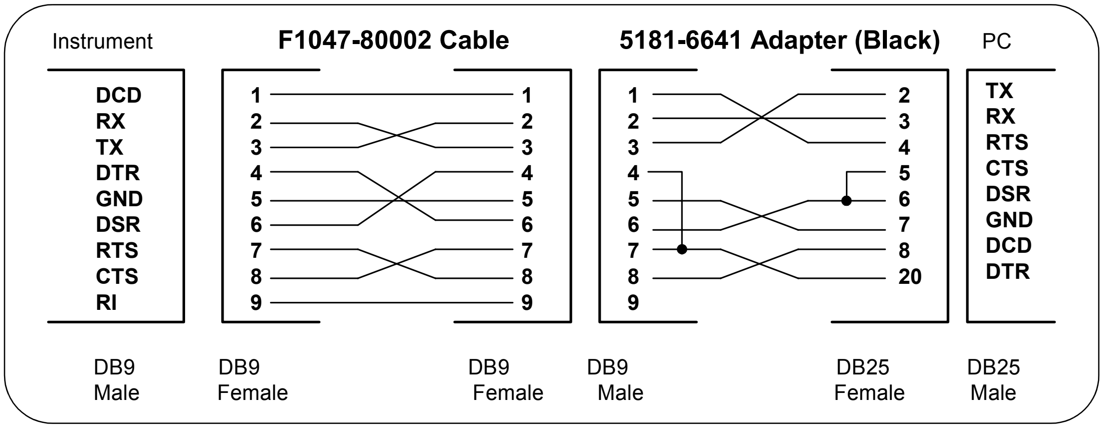


Figure 9 5181-6641 Adapter (Black)

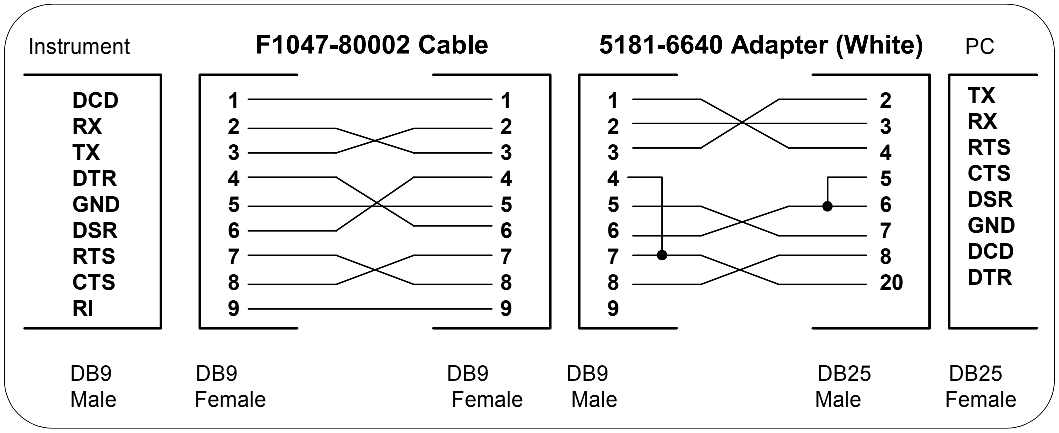


Figure 10 5181-6640 Adapter (White)

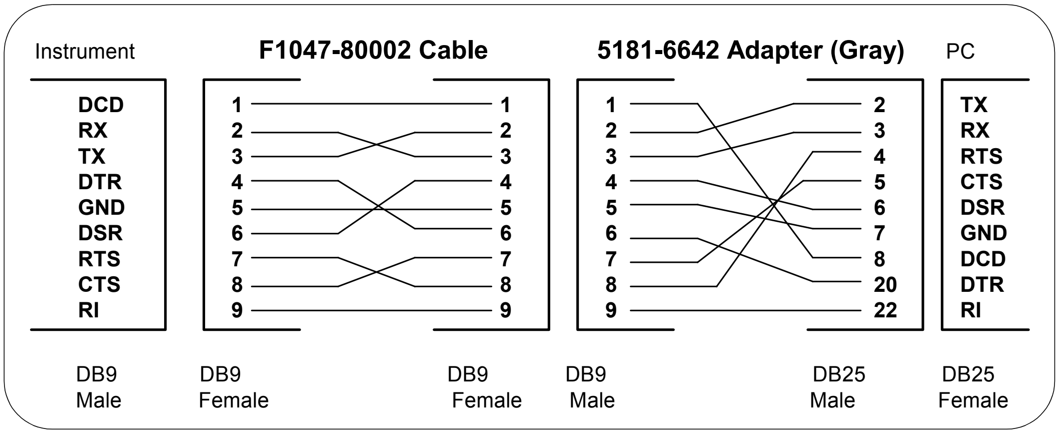


Figure 11 5181-6642 Adapter (Gray)

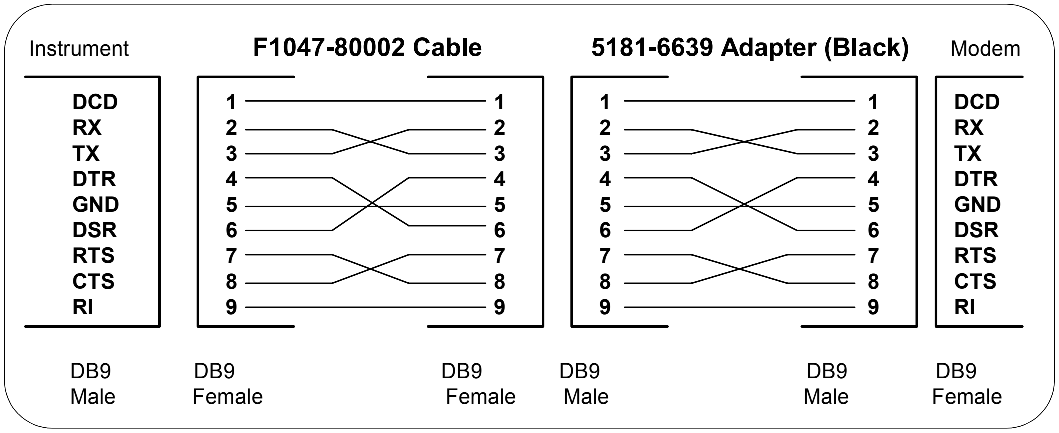


Figure 12 5181-6639 Adapter (Black)

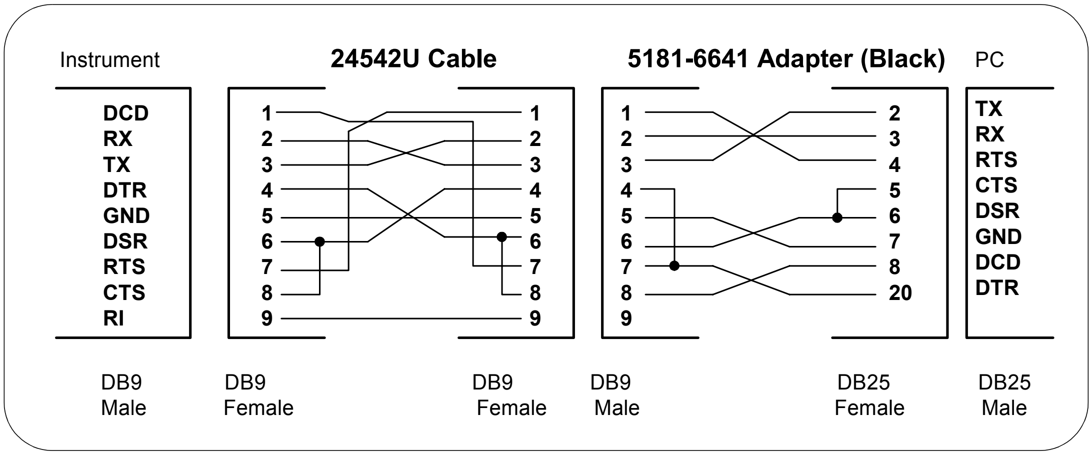


Figure 13 5181-6641 Adapter (Black)

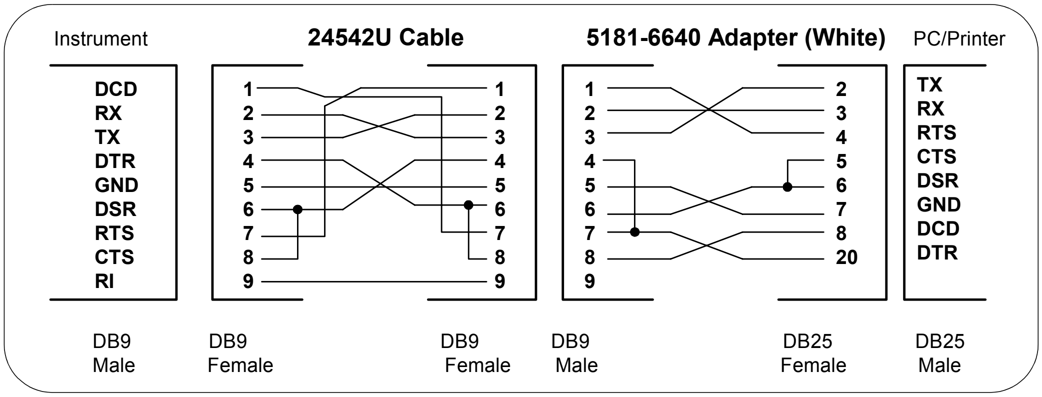


Figure 14 5181-6640 Adapter (White)

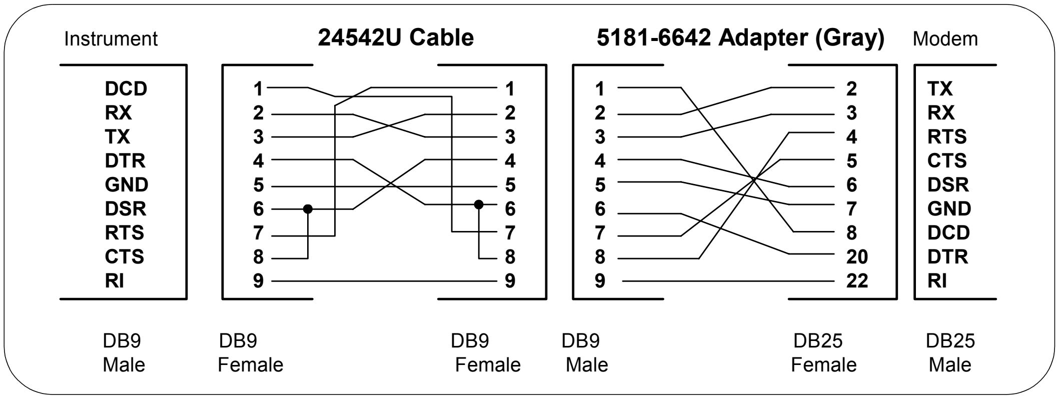


Figure 15 5181-6642 Adapter (Gray)

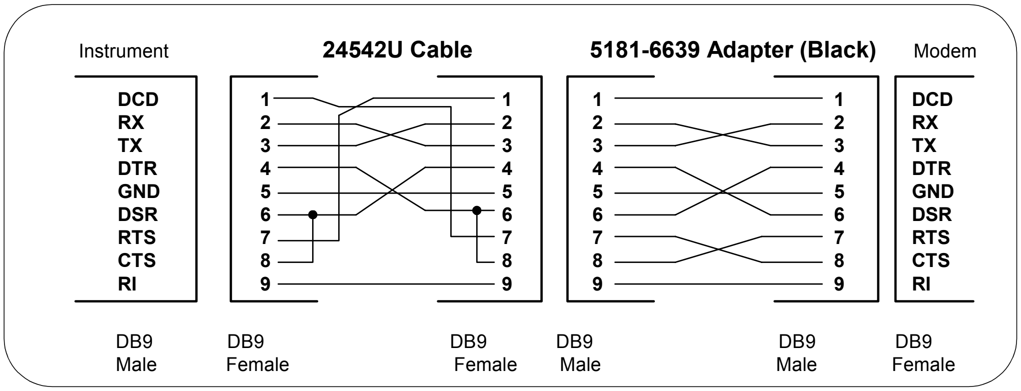


Figure 16 5181-6639 Adapter (Black)

Appendix B: Troubleshooting SICL Programs

Troubleshooting SICL Programs

This chapter 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 GPIO Problems
- Common LAN Problems

SICL Error Codes

When you install a default SICL error handler such as I_ERROR_EXIT or I_ERROR_NOEXIT with an **ionerror** call, a SICL internal error message is logged. To view these messages:

- On Windows 98SE or Windows Me, start the **Message Viewer** utility by clicking the Agilent IO Libraries Control (on the taskbar) and then clicking **Run Message Viewer**. You *must* start the **Message Viewer** utility before you execute a program for error messages to be logged.
- On Windows 2000, XP, or NT, SICL logs internal messages as Windows NT events that you can view by clicking the Agilent IO Libraries Control (on the taskbar) and then clicking **Run Event Viewer**. Both system and application messages can be logged to the **Event Viewer** from SICL. SICL messages are identified by **SICL LOG** or by the driver name (such as **hp341i32** for the GPIB driver).

For C programs, you can use **ionerror** to install a custom error handler. The error handler can call **igeterrstr** with the given error code and the corresponding error message string will be returned. See *Chapter 3 - Programming with SICL* for more information on error handlers. This table summarizes SICL error codes and messages.

Table 60 List of SICL Error Codes

Error Code	Error String	Description
I_ERR_ABORTED	Externally aborted	A SICL call was aborted by external means.
I_ERR_BADADDR	Bad address	The device/interface address passed to **iopen** does not exist. Verify that the interface name is the one assigned with IO Config.
I_ERR_BADCONFIG	Invalid configuration	An invalid configuration was identified when calling **iopen**.
I_ERR_BADFMT	Invalid format	Invalid format string specified for **iprintf** or **iscanf**.
I_ERR_BADID	Invalid INST	The specified **INST** *id* does not have a corresponding **iopen**.
I_ERR_BADMAP	Invalid map request	The **imap** call has an invalid map request.
I_ERR_BUSY	Interface is in use by non-SICL process	The specified interface is busy.
I_ERR_DATA	Data integrity violation	The use of CRC, Checksum, and so forth imply invalid data.
I_ERR_INTERNAL	Internal error occurred	SICL internal error.
I_ERR_INTERRUPT	Process interrupt occurred	A process interrupt (signal) has occurred in your application.
I_ERR_INVLADDR	Invalid address	The address specified in **iopen** is not a valid address (for example, "hpib,57").
I_ERR_IO	Generic I/O error	An I/O error has occurred for this communication session.
I_ERR_LOCKED	Locked by another user	Resource is locked by another session (see **isetlockwait**).
I_ERR_NESTEDIO	Nested I/O	Attempt to call another SICL function when current SICL function has not completed (WIN16). More than one I/O operation is prohibited.
I_ERR_NOCMDR	Commander session is not active or available	Tried to specify a commander session when it is not active, available, or does not exist.

Table 60 List of SICL Error Codes

Error Code	Error String	Description
I_ERR_NOCONN	No connection	Communication session has never been established, or connection to remote has been dropped.
I_ERR_NODEV	Device is not active or available	Tried to specify a device session when it is not active, available, or does not exist.
I_ERR_NOERROR	No Error	No SICL error returned; function return value is zero (0).
I_ERR_NOINTF	Interface is not active	Tried to specify an interface session when it is not active, available, or does not exist.
I_ERR_NOLOCK	Interface not locked	An **iunlock** was specified when device was not locked.
I_ERR_NOPERM	Permission denied	Access rights violated.
I_ERR_NORSRC	Out of resources	No more system resources available.
I_ERR_NOTIMPL	Operation not implemented	Call not supported on this implementation. The request is valid, but not supported on this implementation.
I_ERR_NOTSUPP	Operation not supported	Operation not supported on this implementation.
I_ERR_OS	Generic O.S. error	SICL encountered an operating system error.
I_ERR_OVERFLOW	Arithmetic overflow	Arithmetic overflow. The space allocated for data may be smaller than the data read.
I_ERR_PARAM	Invalid parameter	The constant or parameter passed is not valid for this call.
I_ERR_SYMNAME	Invalid symbolic name	Symbolic name passed to **iopen** not recognized.
I_ERR_SYNTAX	Syntax error	Syntax error occurred parsing address passed to **iopen**. Make sure you have formatted the string properly. White space is not allowed.
I_ERR_TIMEOUT	Timeout occurred	A timeout occurred on the read/write operation. The device may be busy, in a bad state, or you may need a longer timeout value for that device. Check also that you passed the correct address to **iopen**.

Table 60 List of SICL Error Codes

Error Code	Error String	Description
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 61 Windows Errors

Windows 98SE and Windows Me	
Subsequent Execution of SICL Application Fails	If you terminate a program using the Task Manager, or if a program has an abnormal termination, some drivers may not unload from memory. This could cause subsequent attempts to execute the I/O program to fail. To recover from this situation, you must restart (reboot) Windows 98SE/Me.
Windows 2000, XP, and NT	
Program Appears to Hang and Cannot Be Stopped	Check that an **itimeout** value has been set for all SICL sessions in your program. Otherwise, when an instrument does not respond to a SICL read or write, SICL will wait indefinitely in the SICL kernel access routine, preventing the application from being stopped. To stop the application, click the "toaster" button in the upper-left corner of the window and then close the window. After a few seconds, an **End Task** dialog box appears. Press the **End Task** button to stop the application.
Formatted I/O Using `%F` Causes Application Error	Verify `$(`**cvarsdll**`)` is used when compiling the application, and either `$(`**guilibsdll**`)` for Windows applications or `$(`**conlibsdll**`)` for console applications when linking your application. Also, the `%F` format character for **iprintf** only works with languages that use *MSVCRT.DLL*, *MSVCRT20.DLL*, or *MSVCRT40.DLL* for their run-time library. Some versions of Visual C/C++ and Borland C/C++ use their own versions of the run-time library. They cannot share global data with SICL's version of the run-time library and, therefore, cannot use `%F`.

Common RS-232 Problems

Unlike GPIB, special care must be taken to ensure that RS-232 devices are correctly connected to the computer. Verifying the configuration first may save many hours of debugging time.

Table 62 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.
Data Lost During Large Transfers	Check: Flow control setting match Full/half duplex for 3-wire connections Cabling is correct for hardware handshaking

Common GPIO Problems

Because the GPIO interface has such flexibility, most initial problems come from cabling and configuration. There are many configuration fields in the IO Config utility that must be configured for GPIO. For example, no data transfers will work correctly until the handshake mode and polarity have been correctly set. A GPIO cable can have up to 50 wires and you may need to solder your own plug to at least one end. It is important to ensure correct hardware configuration before you begin troubleshooting the software.

If you are porting an existing 98622 application, the hardware task is simplified. The cable connections are the same and many IO Config fields closely approximate 98622 DIP switches. For a new application, an individual with good hardware skills should become familiar with the E2075 cabling and handshake behavior. In either case, you may want to read the *Agilent E2075 GPIO Interface Card Installation Guide*.

Some GPIO-specific reasons for certain SICL errors follow. Many of these errors can also be caused by non-GPIO problems. For example, “Operation not supported” will happen on any interface if you execute **igetintfsess** with an interface ID.

Bad Address (for `iopen`)

This indicates **iopen** did not succeed because the specified address (symbolic name) did not correspond to a correctly configured entry in IO Config. If **iopen** fails, be sure the interface is properly configured. IO Config establishes an entry for the GPIO card in the Windows 98SE/2000/NT registry.

We strongly encourage you to let IO Config handle all registry maintenance for SICL. However, you can edit registry entries manually. If you manually change the registry and enter an improper configuration value, the failed **iopen** may send a diagnostic message to the **Message Viewer** (Windows 98SE/Me) or **Event Viewer** (Windows 2000/XP/NT).

For example:

```
HPe2074: GPIO config, bad read_clk entry
 ISA card in slot #0 NOT INITIALIZED (Invalid
 parameter)
```

In this case, you must correct the configuration data in the registry. The recommended procedure is to use IO Config, remove the incorrect interface name, and create a **Configured Interface** with legal values selected from the IO Config utility's dialog boxes.

Operation Not Supported

The E2075 has several modes. Certain operations are valid in one mode, and not supported in another. Two examples are:

```
igpioctrl(id, I_GPIO_AUX, value);
```

This operation applies only to the Enhanced mode of the data port. Auxiliary control lines do not exist when the interface is in 98622 Compatibility mode.

```
igpioctrl(id, I_GPIO_SET_PCTL, 1);
```

This operation is allowed only in Standard-Handshake mode. When the interface is in Auto-Handshake mode (the default), explicit control of the PCTL line is not possible.

No Device

This error indicates PSTS checks were set for read/write operations and a false state of the PSTS line was detected. Enabling and disabling PSTS checks is done with:

```
igpioctrl(id, I_GPIO_CHK_PSTS, value);
```

If the check seems to be reporting the wrong state of the PSTS line, correct the PSTS polarity bit via the IO Config utility. If the PSTS check is functioning properly and this error is generated, some problem with the cable or the peripheral device is indicated.

Bad Parameter

If the interface is in 16-bit mode, the number of bytes requested in an **iread** or **iwrite** function must be an even number. Although you probably view 16-bit data as words, the syntax of **iread** and **iwrite** requires a length specified as bytes.

Common LAN Problems

NOTE

Both the LAN client and LAN server may log messages to the **Message Viewer** (Windows 98SE/Me) or **Event Viewer** (Windows 2000/XP/NT) under certain conditions, whether or not an error handler has been registered.

General Troubleshooting Techniques

Before SICL over LAN can function, the client must be able to talk to the server over the LAN. You can use the following techniques to determine if the problem is a general network problem or is specific to the LAN software provided with SICL.

Using the `ping` Utility

If the application cannot open a session to the LAN server for SICL, the first diagnostic to try is the **ping** utility. This utility allows you to test general network connectivity between client and server machines.

Using **ping** looks something like the following, where each line after the **Pinging** line is an example of a packet successfully reaching the server.

```
>ping instserv.hp.com

Pinging instserv.hp.com[128.10.0.3] with 32
bytes of data:Reply from 128.10.0.3:bytes=32
  time=10ms TTL=255
Reply from 128.10.0.3:bytes=32 time=10ms
  TTL=255
Reply from 128.10.0.3:bytes=32 time=10ms
  TTL=255
Reply from 128.10.0.3:bytes=32 time=10ms
  TTL=225
```

However, if **ping** cannot reach the host, a message similar to the following is displayed that indicates the client was unable to contact the server. In this case, you should contact your network administrator to determine if there is a LAN problem. When the LAN problem has been corrected, you can retry your SICL application over LAN.

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

LAN Client Problems

iopen Fails - Syntax Error

In this case, **iopen** fails with the error I_ERR_SYNTAX. If using the “lan,net_address” format, ensure that the net_address is a hostname, not an IP address. If you must use an IP address, specify the address using the bracket notation, **lan[128.10.0.3]**, rather than the comma notation **lan,128.10.0.3**.

iopen Fails - Bad Address

An **iopen** fails with the error I_ERR_BADADDR, and the error text is **core connect failed: program not registered**. This indicates the LAN server for SICL has not registered itself on the server machine. This may also be caused by specifying an incorrect hostname. Ensure that the hostname or IP address is correct and, if so, check the LAN server's installation and configuration.

iopen Fails - Unrecognized Symbolic Name

The **iopen** fails with the error I_ERR_SYMNAME, and the error text is **bad hostname, gethostbyname() failed**. This indicates the hostname used in the **iopen** address is unknown to the networking software. Ensure that the hostname is correct and, if so, contact your network administrator to configure your machine to recognize the hostname. The **ping** utility can be used to determine if the hostname is known to your system. If **ping** returns with the error **Bad IP address**, the hostname is not known to the system.

iopen Fails - Timeout

An **iopen** fails with a timeout error. Increase the Client Timeout Delta configuration value via the IO Config utility. See *Chapter 8 - Using SICL with LAN* for more information.

iopen Fails - Other Failures

An **iopen** fails with some error other than those already mentioned. Try the steps at the beginning of this section to see if the client and server can talk to one another over the LAN. If the **ping** and **rpcinfo** procedures work, check any server error logs that may be available for further clues. Check for possible problems such as a lack of resources at the server (memory, number of SICL sessions, etc.).

I/O Operation Times Out

An I/O operation times out even though the timeout being used is infinity. Increase the Server Timeout configuration value via the IO Config utility. Also, ensure the LAN client timeout is large enough if **ilantimeout** is used. See *Chapter 8 - Using SICL with LAN* for more information.

Operation Following a Timed Out Operation Fails

An I/O operation following a previous timeout fails to return or takes longer than expected. Ensure the LAN timeout being used by the system is sufficiently greater than the SICL timeout being used for the session in question. The LAN timeout should be large enough to allow for the network overhead in addition to the time that the I/O operation may take.

If **ilantimeout** is used, you must determine and set the LAN timeout manually. Otherwise, ensure the Client Timeout Delta configuration value is large enough (via the IO Config utility). See *Chapter 8 - Using SICL with LAN* for more information.

iopen Fails or Other Operations Fail Due to Locks

An **iopen** fails due to insufficient resources at the server or I/O operations fail because some other session has the device or interface locked. LAN server connections for SICL from previous clients may not have terminated properly. Consult your server's troubleshooting documentation and follow the instructions for cleaning up any previous server processes.

LAN Server Problems

SICL LAN Application Fails - RPC Error

After starting the LAN server, a SICL LAN application fails and returns a message similar to the following:

```
RPC_PROG_NOT_REGISTERED
```

There is a short (approximately 5 second) delay between starting the LAN server and the LAN server being registered with the Portmapper. Try running the SICL LAN application again.

rpcinfo Does Not List 395180 or 395183

A **rpcinfo** query fails to indicate that program **395180** (SICL LAN Protocol) or **395183** (TCP/IP Instrument Protocol) is available on the server. If you have not yet started the LAN server, do so now. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for details 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 *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on using IO Config to configure interfaces for SICL.

LAN Server Appears "Hung"

The LAN server appears "hung" (possibly due to a long timeout being set by a client on an operation that will never succeed). Login to the LAN server and stop the hung LAN server process. To stop the LAN server, see the *Agilent IO Libraries Installation and Configuration Guide for Windows*.

This action will affect all connected clients, even those that may still be operational. If informational logging has been enabled using the IO Config utility, connected clients can be determined by log entries in the **Message Viewer** (Windows 98SE/Me) or **Event Viewer** (Windows 2000/XP/NT) 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 *Agilent IO Libraries Installation and Configuration Guide for Windows* for details to start and stop the LAN server.

rpcinfo Fails - program 395180 is not available

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

```
rpcinfo: RPC_SYSTEM_ERROR - Connection refused
program 395180 version 1 is not available
```

Ensure that the LAN server program is running on the server.

Mouse "Hung" When Stopping LAN Server

After attempting to stop a LAN server via either **Ctrl+C** or the Windows 98SE/Me/2000/XP/NT x-button (in the upper-right hand corner of the window), the mouse may appear to be "hung." Press any keyboard key and the LAN server will stop and the mouse will again be operational.

Glossary

address

A string uniquely identifying a particular interface or a device on that interface.

bus error

An action that occurs when access to a given address fails, either because no register exists at the given address, or the register at the address refuses to respond.

bus error handler

Programming code executed when a bus error occurs.

commander session

A session that communicates to the controller of this bus.

controller

A computer used to communicate with a remote device such as an instrument. In the communications between the controller and the device, the controller is in charge of, and controls, the flow of communication (that is, does the addressing and/or other bus management).

controller role

A computer acting as a controller communicating with a device.

device

A unit that receives commands from a controller. Typically a device is an instrument but could also be a computer acting in a non-controller role, or another peripheral such as a printer or plotter.

device driver

A segment of software code that communicates with a device. It may either communicate directly with a device by reading and writing registers, or it may communicate through an interface driver.

device session

A session that communicates as a controller specifically with a single device, such as an instrument.

handler

A software routine used to respond to an asynchronous event such as an error or an interrupt.

instrument

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

interface

A connection and communication medium between devices and controllers, including mechanical, electrical, and protocol connections.

interface driver

A software segment that communicates with an interface. It also handles commands used to perform communications on an interface.

interface session

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

interrupt

An asynchronous event requiring attention out of the normal flow of control of a program.

lock

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

logical unit

A logical unit is a number associated with an interface. In SICL, a logical unit uniquely identifies an interface. Each interface on the controller must have a unique logical unit.

mapping

An operation that returns a pointer to a specified section of an address space, and makes the specified range of addresses accessible to the requester.

non-controller role

A computer acting as a device communicating with a controller.

process

An operating system object containing one or more threads of execution that share a data space. A multi-process system is a computer system that allows multiple programs to execute simultaneously, each in a separate process environment. A single-process system is a computer system that allows only a single program to execute at a given point in time.

register

An address location that controls or monitors hardware.

session

An instance of a communications channel with a device, interface, or commander. A session is established when the channel is opened with the **iopen** function and is closed with a corresponding call to **iclose**.

SRQ

Service Request. An asynchronous request (an interrupt) from a remote device indicating that the device requires servicing.

status byte

A byte of information returned from a remote device showing the current state and status of the device.

symbolic name

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

thread

An operating system object that consists of a flow of control within a process. A single process may have multiple threads that each have access to the same data space within the process. However, each thread has its own stack and all threads may execute concurrently with each other (either on multiple processors, or by time-sharing a single processor). Multi-threaded applications are only supported with 32-bit SICL.

Index