HP Standard Instrument Control Library

User's Guide for Windows

Notice

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

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

Warranty Information.

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

Restricted Rights Legend.

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

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Printing History

This is the third edition of the *HP Standard Instrument Control Library User's Guide for Windows*.

April 1994 — First Edition

September 1995 — Second Edition

May 1996 — Third Edition

Contents

1

Introduction

Introduction

Welcome to the *HP Standard Instrument Control Library (SICL) User's Guide for Windows*. This guide explains how to use SICL to develop I/O applications on Microsoft® Windows™ environments. 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 the HP-IB, GPIO, RS-232, and LAN interfaces.

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

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

- Chapter 2 - Getting Started with HP SICL steps you through building and running a simple example program in C/C++ and in Visual BASIC. This is a good place to start if you are a first-time SICL user.
- Chapter 3 - Building an HP SICL Application explains how to build a SICL application in a Windows environment.
- Chapter 4 - Programming with HP SICL provides some detailed example programs and considerations to remember when programming in a Windows environment. You can find information on communications sessions, addressing, error handling, locking, and more.
- Chapter 5 - Using HP SICL with HP-IB describes how to communicate over the HP-IB interface. Example programs are also provided.
- Chapter 6 - Using HP SICL with GPIO describes how to communicate over the GPIO interface. Example programs are also provided.
- Chapter 7 - Using HP SICL with RS-232 describes how to communicate over the RS-232 interface. Example programs are also provided.
- Chapter 8 - Using HP SICL with LAN describes how to communicate over a Local Area Network (LAN). Example programs are also provided.

- Chapter 9 - Troubleshooting Your HP SICL Program describes some of the most common SICL programming problems and provides troubleshooting procedures to help you solve the problems.
- Chapter 10 - More HP SICL Example Programs contains additional example programs to help you develop your SICL applications.

This guide also contains the following appendices:

- Appendix A - HP SICL System Information provides information on SICL software files and system interaction.
- Appendix B - Porting from the HP 82335 Command Library provides tips for moving from the HP-IB Command Library products to SICL.
- Appendix C - Porting to Visual BASIC 4.0 explains how to move SICL applications from earlier versions of Visual BASIC (such as version 3.0) to Visual BASIC version 4.0.

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

SICL is intended for instrument I/O and C/C++ or Visual BASIC programmers who are familiar with the Microsoft Windows 95™, Windows NT®, or Windows 3.1™ environment. This manual does not attempt to teach the C/C++ or Visual BASIC programming languages, nor does it attempt to teach instrument I/O concepts.

If you will be performing the SICL installation and configuration on Windows NT, you must also have system administration privileges on your Windows NT system.

HP SICL Overview

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

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

There is a 32-bit and a 16-bit version of SICL for Windows. Note that you can use one or both versions of SICL on your 32-bit computer when running the Windows 95 environment. The following two tables summarize the support for the 32-bit and 16-bit versions of SICL, including which Microsoft Windows environments, I/O interfaces, and programming languages are supported for each version.

Support for 32-bit SICL

Environment	Interfaces	Programming Languages
Windows 95	HP-IB, VXI[1], RS-232, GPIO, LAN	C, C++, Visual BASIC
Windows NT	HP-IB, RS-232, GPIO, LAN	C, C++, Visual BASIC

Support for 16-bit SICL

Environment	Interfaces	Programming Languages
Windows 95	HP-IB, VXI[1], RS-232, GPIO	C, C++, Visual BASIC
Windows 3.1	HP-IB, VXI[1], RS-232	C, C++, Visual BASIC

[1] SICL for the VXI interface is shipped with the HP VXI Embedded PC Controller and VXLink products.

NOTE

This edition of this manual supports and shows how to program SICL applications in Visual BASIC version 4.0 or later.

If you have SICL applications written in an earlier Visual BASIC version than version 4.0 (for example, version 3.0), you can easily port your SICL applications to Visual BASIC version 4.0. See Appendix C, "Porting to Visual BASIC 4.0," in this manual.

HP SICL Features

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

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

Other Documentation

The following documentation is also helpful when using SICL:

- *HP I/O Libraries Installation and Configuration Guide for Windows* explains how to install and configure the HP Virtual Instrument Software Architecture (VISA) library, HP VISA Transition Library (VTL), and SICL on a Microsoft Windows environment.
- *HP SICL Reference Manual* provides the function syntax and description of each SICL function.
- *HP SICL Quick Reference Guide for C Programmers* helps you find SICL function syntax information quickly if you are programming in C/C++.
- *HP SICL Quick Reference Guide for Visual BASIC Programmers* helps you find SICL function syntax information quickly if you are programming in Visual BASIC.
- *HP SICL Online Help* is provided in the form of Windows Help.
- *HP SICL Example Programs* are provided in the `C\SAMPLES` (for C/C++) subdirectory and in the `VB\SAMPLES` subdirectory (for Visual BASIC) under the base directory where SICL is installed (for example, under the `C:\SICL` base directory, if the default installation directory is used). These examples are designed to help you develop your SICL applications more easily.

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

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

2

Getting Started with HP SICL

Getting Started with HP SICL

This chapter will help you to get started programming with SICL. This chapter steps through a simple example to let you verify your configuration and introduce you to some of SICL's basic features.

This chapter is divided into two sections: the first is for C programmers, and the second is for Visual BASIC programmers. Please go to the appropriate section depending on whether you will use SICL with the C/C++ programming language, or SICL with the Visual BASIC programming language.

You may also want to look through the *HP SICL Reference Manual* to familiarize yourself with SICL's functionality. Note that this reference information is also available as online help. To get the reference information online, simply double-click on the `Help` icon either in the `HP I/O Libraries` program group for Windows 95 or Windows NT, or in the `HP SICL` program group for Windows 3.1.

Getting Started Using C

In this section, you will first review a simple example program called `IDN` that queries an HP-IB instrument for its identification string. This example either uses the QuickWin or EasyWin feature of Microsoft and Borland C compilers for WIN16 programs (that is, for 16-bit SICL programs on the Windows 95 or Windows 3.1 environment), or it builds a console application for WIN32 programs (that is, for 32-bit SICL programs on the Windows 95 or Windows NT environment). Once you have reviewed the program, you will then learn how to compile and run the example program.

Reviewing an HP SICL Program

All files used to develop SICL applications in C or C++ are located in the `C` subdirectory of the base SICL directory (for example, `C:\SICL\C` if you installed SICL in the default location). Sample C/C++ programs are located in the `C\SAMPLES` subdirectory of the base SICL directory (for example, `C:\SICL\C\SAMPLES`). Each sample program subdirectory contains makefiles or project files that you can use to build each sample C program. Note that 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 SICL base directory (for example, `C:\SICL\C\SAMPLES\IDN`). This subdirectory contains the source program, `IDN.C`.

The source file `IDN.C` is listed on the following pages. An explanation of the various function calls in the example is provided directly after the program listing for your review.

```
////////////////////////////////////////////////////////////////////
//
// The following simple demonstration program uses the Standard
// Instrument Control Library to query an HP-IB instrument for
// an identification string and then prints the result.
//
// Edit the DEVICE_ADDRESS line below to specify the address of the
// device you want to talk to.  For example:
//
//     hpib7,0    - refers to an HP-IB device at bus address 0
//                  connected to an interface named "hpib7" by the
//                  I/O Config utility.
//
//     hpib7,9,0  - refers to an HP-IB device at bus address 9,
//                  secondary address 0, connected to an interface
//                  named "hpib7" by the I/O Config utility.
//
// Note that this program is meant to be built either as a WIN16
// QuickWin or EasyWin program on Windows 95 or Windows 3.1, or as
// a WIN32 console application on Windows 95 or Windows NT.  Also
// note that WIN16 programs must be compiled with the Large memory
// model.
//
////////////////////////////////////////////////////////////////////

#include <stdio.h>  // for printf()
#include "sicl.h"  // Standard Instrument Control Library routines

#define DEVICE_ADDRESS "hpib7,0"   // Modify this line to match your setup

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

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

   // Install a default SICL error handler that logs an error message and
   // exits.  On Windows 95 and Windows 3.1, view messages with the SICL
   // Message Viewer, and on Windows NT use the Windows NT 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
    // in a known state.
    iprintf(id, "*RST\n");

    // Write the *IDN? string and send an EOI indicator, then read
    // the response into buf.
    // For WIN16 programs, this will only work with the Large memory model
    // since ipromptf expects to receive far pointers to the format strings.

    ipromptf(id, "*IDN?\n", "%t", buf);
    printf("%s\n", buf);

    iclose(id);

    // For WIN16 programs, call _siclcleanup before exiting to release
    // resources allocated by SICL for this application.  This call is a
    // no-op for WIN32 programs.
    _siclcleanup();
}
```

The SICL example C program includes the following:

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

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

`ionerror`

The first SICL call, `ionerror`, installs a default error handling routine that is automatically called if any of the subsequent SICL calls result in an error. `I_ERROR_EXIT` specifies a built-in error handler that will print out a message about the error and then exit the program. If desired, a custom error handling routine could be specified instead. On Windows 95 and Windows 3.1, the error messages may be viewed by executing the **`Message Viewer`** utility either in the **`HP I/O Libraries`** program group for Windows 95, or in the **`HP SICL`** program group for Windows 3.1. On Windows NT, these messages may be viewed with the Windows NT **`Event Viewer`** utility in the **`Administrative Tools`** group.

`iopen`

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

You may wish to modify the program to set the interface name and instrument address to those applicable for your setup. Refer to the section titled "Opening a Communications Session" in Chapter 4, "Programming with HP SICL," for a complete description of how to use SICL's addressing capabilities.

`itimeout`

Next, `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` is used to query 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 HP-IB bus management necessary to perform these reads and writes to instrument.

`iclose` and `_siclcleanup`

The `iclose` function closes the device session to this instrument (`id` is no longer valid after this point). Finally, for WIN16 programs, a call to `_siclcleanup` tells Windows 95 or Windows 3.1 that the WIN16 program is done and that the SICL I/O resources are no longer needed. WIN32 programs on Windows 95 or Windows NT do not require the `_siclcleanup` call.

Refer to the *HP SICL Reference Manual* or SICL online `Help` for more detailed information on these SICL function calls and to learn about all of the functions provided by SICL.

Compiling an HP SICL Program

In this subsection, you will learn how to compile the `IDN` example.

The `C\SAMPLES\IDN` subdirectory (under the SICL base directory) contains a number of files to help you build the example with specific compilers. You will have a subset of the following files depending on which Windows environment you are using.

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

Follow these steps to compile the `IDN` example program:

1. Connect an instrument to your HP 82341 or 82335 HP-IB interface that is compatible with IEEE 488.2 directives.
2. Change directories to the location of the example (for example, `CD \SICL\C\SAMPLES\IDN`).

3. The program assumes that the HP-IB interface name is `hpib7` (set using `I/O Config`) and that the instrument is at bus address `0`. If necessary, use an editor to modify the interface name and instrument address on the `DEVICE_ADDRESS` definition line in the `IDN.C` source file.
4. Compile the program:
   - If you use the command line interface for your compiler (on Windows 3.1 only), compile the program using the makefile from the command prompt as follows.
     - For Borland compilers, type: `MAKE BCIDN.MAK`.
     - For Microsoft compilers, type: `NMAKE MSCIDN.MAK`.

     Now go on to the next subsection, "Running an HP SICL Program."
   - If you use the Windows interface for your compiler, select and load the appropriate project or makefile. Then compile the program as follows.
     - For Borland compilers, use `Project | Open Project`. Then select `Project | Build All`.
     - For Microsoft compilers, use `Project | Open`. Then set the include file path by selecting `Options | Directories`. In the Include File Path box, add a semicolon followed by the full path to the `C` subdirectory (for example, `C:\SICL\C` if you installed SICL in the default location). Then select `Project | Re-build All`.

Running an HP SICL Program

To run the `IDN` example program, do the following. Note that if you are using Windows 95 or Windows NT, you should execute the program from a console command prompt.

- If you use the command line interface (Windows 3.1 only):

  Select **`File | Run`** from the Windows Program Manager menu. (For example, type `C:\SICL\C\SAMPLES\IDN\IDN` in the input box. Then click on `OK`.)

- If you use the Windows interface:
  - For Borland, select **`Run | Run`**.
  - For Microsoft, selet **`Project | Execute`** or **`Run | Go`**.

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

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

If the program does not run, refer to the message logger for a list of run-time errors, and see Chapter 9, "Troubleshooting Your HP SICL Program," for assistance in correcting the problem.

Where to Go Next

Now that you understand some basics of programming with SICL, continue on to Chapter 3, "Building an HP SICL Application," and Chapter 4, "Programming with HP SICL." Chapter 4 provides detailed example programs and some considerations for programming in the Windows environment. It also contains information on communications sessions, addressing, error handling, and so forth.

Additionally, you should look at the chapter(s) that describe how to use SICL with your particular interface(s):

- Chapter 5 - "Using HP SICL with HP-IB"
- Chapter 6 - "Using HP SICL with GPIO"
- Chapter 7 - "Using HP SICL with RS-232"
- Chapter 8 - "Using HP SICL with LAN"

You might also want to familiarize yourself with all the SICL functions, which are defined in the *HP SICL Reference Manual* and in the reference information that is provided in the SICL online `Help`.

If you have any problems, see Chapter 9 in this guide, "Troubleshooting Your HP SICL Program."

In this section, you will first review a simple example program called `IOCMD` that allows you to send commands to an instrument. This example is written in Microsoft Visual BASIC. Once you have reviewed the program, you will then learn how to run the example program.

Reviewing an HP SICL Program

All files used to develop SICL applications in Visual BASIC are located in the `VB` subdirectory of the base SICL directory (for example, `C:\SICL\VB` if you installed SICL in the default location). Sample Visual BASIC programs are located in the `VB\SAMPLES` subdirectory of the base SICL directory (for example, `C:\SICL\VB\SAMPLES`). Each sample program subdirectory contains the project files that you can use to load and run each sample Visual BASIC program.

The `IOCMD` example files are located in the `VB\SAMPLES\IOCMD` subdirectory under the SICL base directory (for example, `C:\SICL\VB\SAMPLES\IOCMD`). The main source file, `IOCMD.FRM`, is listed on the following pages. An explanation of the various function calls in the example is provided directly after the program listing for your review.

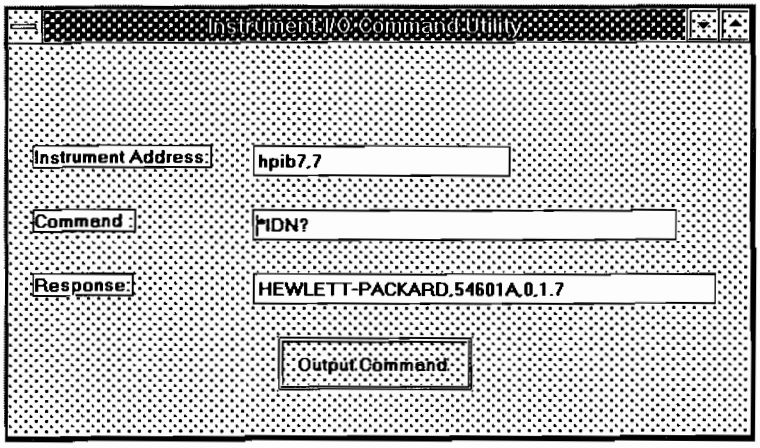
```
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'  This routine uses the Standard Instrument Control
'  Library to send commands to an instrument.  The address
'  of the instrument is obtained from a Text box named
'  txtInstAddr.  If the command is a SCPI query command
'  then the response to the command will be read and
'  displayed in the txtResponse Text box.
'
'  Note that any SICL errors that occur are displayed in
'  the txtResponse Text box.
'
'  This routine is called each time the cmdOutputCmd Command
'  button is clicked.
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Sub cmdOutputCmd_Click ()
   Dim id As Integer                ' device session id
   Dim readbuf As String * 128      ' buffer used for iread
   Dim commandstr As String * 128   ' command passed to instrument
   Dim index As Integer             ' used to parse SCPI error message
   Dim nargs As Integer             ' # args converted by format string

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

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

'  Clear the response string in the txtResponse TextBox.
   txtResponse.Text = ""

'  Open a device session using the device address contained in
'  the Text field of the txtInstAddr TextBox.
   id = iopen(txtInstAddr.Text)

'  Set the I/O timeout value for this session to 1 second.
   Call itimeout(id, 1000)

'  Clear the error/event queue for the instrument.  This allows
'  us to query the instrument after sending a command to see if
'  the command was accepted.
   nargs = ivprintf(id, "*CLS" + Chr$(10), 0&)

'  Write the command to the instrument terminated by a linefeed.
   commandstr = txtCommand.Text + Chr$(10)
   nargs = ivprintf(id, commandstr, 0&)
```

```
'  If the command is a SCPI query command ending in '?',
'  then read and display the response to the command.
   If InStr(txtCommand.Text, "?") Then
      nargs = ivscanf(id, "%128t", readbuf)

      '  Strip out returns and line feeds from the response string.
      readbuf = strip_crlf(readbuf)

      '  Display the response string in the Text field of the
      '  txtResponse TextBox.
      txtResponse.Text = readbuf
   End If

'  Query the instrument to see if the command was accepted
   nargs = ivprintf(id, "SYST:ERR?" + Chr$(10), 0&)
   nargs = ivscanf(id, "%128t", readbuf)

'  Strip out returns and line feeds from the response string.  Note
'  that strip_crlf is a utility routine defined in SICL.BAS and SICL4.BAS.
   readbuf = strip_crlf(readbuf)

'  The SCPI error # is separated by the error message by a ',' character
   index = InStr(readbuf, ",")
   If index <> 0 Then
      If Val(Left$(readbuf, index - 1)) <> 0 Then
         txtResponse.Text = "SCPI Error " + readbuf
      End If
   Else
'     handle non-SCPI errors
      txtResponse.Text = "Error " + readbuf
   End If

'  Close the device session.
   Call iclose(id)

'  Enable the button used to initiate I/O
   cmdOutputCmd.Enabled = True
   Exit Sub

ErrorHandler:

'  Display the error message in the txtResponse TextBox.
   txtResponse.Text = "*** Error : " + Error$

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

```
'  Enable the button used to initiate I/O
   cmdOutputCmd.Enabled = True

   Exit Sub

End Sub

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'  The following routine is called when the application's
'  Start Up form is unloaded.  It calls siclcleanup to
'  release resources allocated by SICL for this
'  application.
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Sub Form_Unload (Cancel As Integer)
   Call siclcleanup    ' Tell SICL to clean up for this task
End Sub
```

The SICL example Visual BASIC program includes the following:

`cmdOutputCmd_Click` Subroutine that gets called when the `cmdOutputCmd` command button is pressed. The command button is labeled `Output Command` (see the following figure).



`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 will be called any time an error occurs during the processing of the `cmdOutputCmd_Click` procedure. Note that SICL errors are handled in the same way that Visual BASIC errors are handled with the `On Error` statement.

`cmdOutputCmd.Enabled` Notice how the button that causes the `cmdOutputCmd_Click` routine to be called is disabled when code is executing inside `cmdOutputCmd_Click`. This is good programming style.

iopen

Next, an **iopen** call is made. The instrument address string is obtained from the Text box control labeled **Instrument Address:** (see the previous figure). This Text box is named **txtInstAddr**, and the instrument address string is taken from its Text property. Note that, in this example, the default contents of **txtInstAddr.Text** is **"hpib7,7"** (the default was set in the Properties box for the **txtInstAddr** control.) The interface name **"hpib7"** is the name given to the interface during execution of the **I/O Config** utility. The bus (primary) address of the instrument follows, in this case **7**, and is typically set with switches on the instrument, or from the front panel of the instrument.

You can specify another address by simply typing it in to the Text box labeled **Instrument Address**. Refer to the section on "Creating a Communications Session" in Chapter 4 for a complete discussion of how to use SICL's addressing capabilities.

The **iopen** call creates a SICL device session, for communication to this specific device (or instrument), with a unique identifier returned. This identifier is used with subsequent SICL calls to indicate which device is being addressed. If only the interface name had been passed to **iopen**, this would have created an interface session instead. In general, device session programming is recommended over interface session programming. This keeps your program from having to understand the intrinsics of the HP-IB bus.

itimeout

Next, **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.

`ivprintf` The first `ivprintf` function is called to send the `*CLS` command to the instrument to clear its error message queue.

The second `ivprintf` function is called to write a command string to the instrument. The command string is obtained from the Text box control labeled `Command:` (see the previous figure). This Text box is named `txtCommand`, and the command string is taken from its Text property. Note that in this example, the default contents of `txtCommand.Text` is `*IDN?` (the default was set in the properties box for the `txtCommand` control.) The `*IDN?` command queries an instrument for its identification string.

In both calls to the `ivprintf` function, a NULL pointer is passed as the last argument. This is because no argument conversion characters were specified in the format string passed as the second argument to `ivprintf`. Note how `0&` is used to specify a NULL pointer in Visual BASIC.

`InStr` The Visual BASIC `InStr` function is called to identify command strings with appended `?` characters. If the command has a `?` character appended to it, then it is a SCPI query command.

`ivscanf` The `ivscanf` function is called to read the response to a SCPI query command. The `"%128t"` format string passed as the second parameter specifies to read up to 128 characters from a device into a string until an END indicator is read. The third parameter is a string used to store the instrument response string.

Note how `readbuf` is declared to be a fixed length string of 128 bytes. Because only 128 bytes total can be read in the format string, `readbuf` will not be overflowed.

`strip_crlf` The `strip_crlf` function is called to strip out carriage returns and line feeds from the response string. Note that `strip_crlf` is a utility routine defined in `SICL.BAS` and `SICL4.BAS`.

`ivprintf` The `ivprintf` function is called to write the `"SYST:ERR?"` command to the instrument. This command queries the instrument to see if the previous command sent to it was accepted.

`ivscanf` The `ivscanf` function is called to read in the response to the `"SYST:ERR?"`.

`strip_crlf`	The `strip_crlf` function is called to strip out carriage returns and line feeds from the response string. Note that `strip_crlf` is a utility routine defined in `SICL.BAS` and `SICL4.BAS`.
`InStr`	The Visual BASIC `InStr` function is called to get the portion of the response string that contains the SCPI error number. The error number is followed by a "," character.
`iclose`	The `iclose` function closes the device session to this instrument (`id` is no longer valid after this point).
`cmdOutputCmd.Enabled`	Notice how the button that causes the `cmdOutputCmd_Click` routine to be called is re-enabled when execution inside `cmdOutputCmd_Click` is finished. This allows more commands to be sent.
`Exit Sub`	This Visual BASIC statement causes the `cmdOutputCmd_Click` subroutine to be exited after normal processing has completed.
`ErrorHandler:`	This label specifies the beginning of the error handler that was installed for the `cmdOutputCmd_Click` subroutine. This handler gets called whenever a run-time error occurs.
`Error$`	This Visual BASIC function is called to get the error message for the error.
`iclose`	The `iclose` function is called to close the SICL session if a session was successfully opened previously (if `id` is not equal to 0).
`cmdOutputCmd.Enabled`	Re-enable the button that causes the `cmdOutputCmd_Click` routine to be called. This allows more commands to be sent.
`Exit Sub`	This Visual BASIC statement causes the `cmdOutputCmd_Click` subroutine to be exited after processing an error in the subroutine's error handler.
`Form_Unload`	This subroutine is called when the Start Up Form is unloaded. This occurs right before a Visual BASIC program exits. This is the appropriate place to call `siclcleanup`.

`siclcleanup` The `siclcleanup` call is used by WIN16 programs to indicate that the program is done and that the SICL I/O resources are no longer needed.

Loading and Running an HP SICL Program

In this subsection, you will learn how to load and run the `IOCMD` Visual BASIC example program.

The `VB\SAMPLES\IOCMD` subdirectory (under the SICL base directory) contains the following files:

`IOCMD.FRM` Visual BASIC source for the `IOCMD` example program.

`IOCMD.MAK` Visual BASIC project file for the `IOCMD` example program.

Follow these steps to load and run the `IOCMD` sample program:

1. Connect an instrument to your interface that is compatible with Standard Commands for Programmable Instruments (SCPI) directives.
2. Run Visual BASIC.
3. Open the project file `IOCMD.MAK` by selecting `File | Open Project` from the Visual BASIC menu.
4. Run the program by pressing (F5), or by pressing the `Run` button on the Visual BASIC Toolbar.
5. Type in the address of the instrument in the Text box labeled `Interface Address`. Note that the default address that appears in this Text box is `"hpib7,7"`. This refers to an HP-IB instrument at bus address 7 connected to the interface named `hpib7`. You should type in the interface name you assigned to your interface (with the `I/O Config` utility) and the bus address of your instrument (if applicable). Make sure to separate the interface name and bus address with a comma, and do not include space characters.

6. Type the command you want to send in the Text box labeled **Command:**. Note that the default command in this Text box is ***IDN?**. This command requests an identification string for the instrument.
7. Press the **Output Command** button to send the command to the instrument at the specified address.

Note that after performing the previous steps, you can create a standalone executable (**.EXE**) version of this program by selecting **File | Make EXE File** from the Visual BASIC menu.

Where to Go Next

Now that you understand the basics of programming with SICL, continue on to Chapter 3, "Building an HP SICL Application," and Chapter 4, "Programming with HP SICL." Chapter 4 provides detailed example programs and some considerations for programming in the Windows environment. It also contains information on communications sessions, addressing, error handling, and so forth.

Additionally, you should look at the chapter(s) that describe how to use SICL with your particular interface(s):

- Chapter 5 - "Using HP SICL with HP-IB"
- Chapter 6 - "Using HP SICL with GPIO"
- Chapter 7 - "Using HP SICL with RS-232"
- Chapter 8 - "Using HP SICL with LAN"

You might also want to familiarize yourself with all the SICL functions, which are defined in the *HP SICL Reference Manual* and in the reference information that is provided in the SICL online **Help**.

If you have any problems, see Chapter 9 in this guide, "Troubleshooting Your HP SICL Program."

3

Building an HP SICL Application

Building an HP SICL Application

This chapter explains how to build a SICL application in a Windows environment. This chapter contains the following sections:

- Including the HP SICL Declaration File
- Memory Models for 16-bit Windows Applications
- Libraries for C Applications and DLLs
- Compiling and Linking C Applications
- Loading and Running Visual BASIC Applications
- Thread Support for 32-bit Windows Applications
- Avoiding Nested I/O in 16-bit Windows Applications
- Application Cleanup

Including the HP SICL Declaration File

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

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

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

Memory Models for 16-bit Windows Applications

We strongly recommend that you use the Large memory model when designing WIN16 applications that call SICL functions. This is because SICL requires all pointer parameters to be "far" pointers. Most SICL function prototypes in the `sicl.h` header file explicitly declare all pointer parameters to be far. However, there is no way to declare pointer types for functions that take a variable number of arguments (such as SICL's formatted I/O functions), and your compiler will not be able to properly check or cast types for these functions.

Libraries for C Applications and DLLs

32-bit Windows

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 SICL base directory (for example, `C:\SICL\C` if you installed SICL in the default location). You may wish 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.

16-bit Windows

All WIN16 applications that use SICL must link to the `SICL16.LIB` import library and to one additional import library that is compiler-dependent. Selecting the proper library depends on the compiler you are using and whether you are calling SICL functions from another dynamic link library (DLL), or from an application program.

`MSAPP16.LIB` Link to this library if you are developing an application with a Microsoft compiler.

`BCAPP16.LIB` Link to this library if you are developing an application with a Borland compiler.

`MSDLL16.LIB` Link to this library if you are developing a DLL with a Microsoft compiler.

`BCDLL16.LIB` Link to this library if you are developing a DLL with a Borland compiler.

All SICL libraries are located in the `C` directory under the SICL base directory (for example, `C:\SICL\C` if you installed SICL in the default location).

32-bit Windows

The following is a summary of important compiler-specific considerations for several C/C++ compiler products when developing WIN32 applications.

For Microsoft Visual C++ compilers:

- Select **Project | Settings** or **Build | Settings** from the menu (depending on the version of your compiler). Click on 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 on **OK** to close the dialog box.
- Select **Project | Settings** from the menu. Click on the **Link** button. Then add **sicl32.lib** to the **Object/Library Modules** list box. Click on **OK** to close the dialog box.
- You may wish to add the SICL **C** directory (for example, **C:\SICL\C**) to the include file and library file search paths. They are set by selecting **Tools | Options** from the menu and clicking on the **Directories** button. Then do the following:
  - To set the include file path, select **Include Files** from the **Show Directories for:** list box. Then click on the **Add** button and type in **C:\SICL\C**. Click on **OK**.
  - To set the library file path, select **Library Files** from the **Show Directories for:** list box. Then click on the **Add** button and type in **C:\SICL\C**. Click on **OK**.

For Borland C++ version 4.0 compilers:

- 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:\SICL\C` if SICL is installed in the default location).
- Edit the `BCC32.CFG` and `TLINK32.CFG` files, which are located in the `BIN` subdirectory of the Borland C installation directory.
  - Add the following line to `BCC32.CFG` so the compiler can find the `sicl.h` file:

    `-IC:\`*SICL_base_dir*`\C`

    where *SICL_base_dir* is the SICL base directory on your system.
  - Add the following line to both files so the compiler and linker can find `BCSICL32.LIB`:

    `-LC:\`*SICL_base_dir*`\C`

    where *SICL_base_dir* is the SICL base directory on your system.
- As an example, to create `MYPROG.EXE` from `MYPROG.C`, you would type:

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

16-bit Windows

The following is a summary of important compiler-specific considerations for several C/C++ compiler products when developing WIN16 applications.

For Microsoft compilers on Windows 3.1 only, such as Microsoft C 7.0 or SDK compilers:

- Make sure the Large memory model is selected using `/AL`.
- Be sure to compile with an option that adds prolog code for exported functions (`/GA` or `/Gsw` for applications, `/GD` for DLLs). This causes the application's data segment to be loaded correctly at the beginning of an exported function. It is also required for SICL error and interrupt handlers to work correctly.
- You may wish to add the SICL `C` directory (for example, `C:\SICL\C`) to the include file and library file search paths. These are typically set using the `LIB` and `INCLUDE` environment variables in the `AUTOEXEC.BAT` file in the root directory. Otherwise, the library and include files and paths should be explicitly specified in the makefile.

For the Microsoft Visual C++ compiler:

- To set the memory model, do the following:
  1. Select `Options | Project`.
  2. Click on the `Compiler` button, then select `Memory Model` from the `Category` list box.
  3. Click on the `Model` list arrow to display the model options, and select `Large`.
  4. Click on `OK` to close the `Compiler` dialog box.
- You may wish to add the SICL `C` directory (for example, `C:\SICL\C`) to the include file and library file search paths. They are set under the `Options | Directories` menu selection. Otherwise, the library and include files and paths should be explicitly specified in the project file.

For Borland C 4.0 compilers:

- Make sure the Large memory model is selected:

  1. Select **Options | Project**.
  2. Double-click on **16-bit Compiler** in the **Topics** list box, then click on **Memory Model**.
  3. Click on the radio button next to **Large** in the **Mixed Model Override** box.
  4. Click on **OK** to close the dialog box.

  You can do this from the command line environment by specifying the **/ml** option to the compiler.

- The Borland C linker defaults to being case-insensitive when resolving references. To link to the SICL libraries, you will need to tell the linker to be case-sensitive.

  To do this from Borland's Integrated Environment:

  1. Select **Options | Project**.
  2. Double-click on **Linker** in the **Topics** list box, then click on **General**.
  3. Click on the checkbox next to **Case Sensitive Exports and Imports**.
  4. Click on **OK** to close the dialog box.

  You can do this from the command line environment by specifying the **/C** option to TLINK.

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. Once you have opened the application's project file, you can run the application by pressing either F5 or the **Run** button on the Visual BASIC Toolbar.

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

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 a few important points to remember:

- 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 wish to make use of the SICL session locking functions (**ilock** and **iunlock**) to help coordinate common instrument accesses from more than one thread.

Also see the "LAN Client and Threads" section in Chapter 8, "Using HP SICL with LAN," for thread information specific to the use of SICL with LAN.

Avoiding Nested I/O in 16-bit Windows Applications

In WIN16 applications, the `I_ERR_NESTED_IO` error is generated by SICL whenever an attempt is made to call a SICL function before a previous call to another SICL function is complete. This error can occur in event-driven WIN16 programs where SICL functions are called in response to events such as menu selections or button clicks.

To avoid this problem, you should disable menu items, buttons, or other controls that cause SICL calls to be made before previous SICL function calls are complete. Note that all of the sample programs that make SICL calls in response to events do this. In particular, the oscilloscope example in Chapter 10 shows one design method to prevent this WIN16 problem.

Application Cleanup

16-bit Windows and C

NOTE

WIN32 SICL applications on Windows 95 or Windows NT do *not* require the **_siclcleanup** call.

SICL has a special function, **_siclcleanup()**, to ensure that Windows performs the necessary clean-up required when a WIN16 SICL program written in C completes execution. Each WIN16 SICL application written in C should call **_siclcleanup()** before exiting or posting a **WM_QUIT** message in order to release resources allocated for the application by the SICL library. Without this call, you may experience difficulty in executing your application, especially from within debuggers.

Note that the **I_ERROR_EXIT** handler calls **_siclcleanup()** automatically before it exits.

16-bit Windows and Visual BASIC

SICL has a special function, **siclcleanup**, to ensure that Windows performs the necessary cleanup required when a 16-bit SICL program written in Visual BASIC completes execution. Each 16-bit SICL application written in Visual BASIC on Windows 95 or Windows 3.1 should call **siclcleanup** before exiting.

The best place to call **siclcleanup** is in the **Form_Unload** routine of the Start Up form in a Visual BASIC program. This is where **siclcleanup** is called in all of the SICL example programs that are written in Visual BASIC throughout this manual.

4

Programming with HP SICL

Programming with HP SICL

This chapter first describes what you need to know to build a SICL application. Then some of the basic features of SICL, such as formatted I/O, error handling, and locking are described.

Detailed example programs are also provided to help you develop your SICL applications more easily. Copies of the example programs are located in the `C\SAMPLES\MISC` subdirectory (for C/C++) and in the `VB\SAMPLES\MISC` subdirectory (for Visual BASIC) under the SICL base directory (for example, under the `C:\SICL` base directory, if SICL was installed in the default location).

This chapter contains the following sections:

- Opening a Communications Session
- Sending I/O Commands
- Handling Asynchronous Events
- Logging Error Messages
- Using Error Handlers
- Using Locks

For specific details on the SICL functions, see the *HP SICL Reference Manual* or the SICL online `Help`.

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.

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 examples.

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 worrying about the type of interface to which it 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 particular device logical address in the *addr* parameter of the `iopen` function. The interface logical unit and symbolic name are set by running the `I/O Config` utility either from the `HP I/O Libraries` program group for Windows 95 or Windows NT, or from the `HP SICL` program group for Windows 3.1. See Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows* for information on the `I/O Config` utility.

The logical unit is an integer corresponding to the interface. The device address generally consists of an integer that corresponds to the device's bus address. It may also include a secondary address which is an integer.

NOTE

Secondary addressing is not supported on RS-232 interfaces.

The following are valid device addresses:

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

The following are examples of opening a device session with the HP-IB device at address 23.

C example:

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

Visual BASIC example:

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

More on addressing specific devices can be found in the interface-specific chapter (for example, "Using HP SICL with HP-IB") later in this manual.

Interface Sessions

An interface session allows direct, low-level control of the specified interface. There is a full set of interface-specific SICL functions for programming features that are specific to a particular interface type (GPIB, Serial, and so forth). This gives you full control of the activities on a given interface, but does make for 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 `I/O Config` utility either from the `HP I/O Libraries` program group for Windows 95 or Windows NT, or from the `HP SICL` program group for Windows 3.1. See Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows* for information on the `I/O Config` utility.

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

The following are valid interface addresses:

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

The following 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")
```

More on addressing specific interfaces can be found in the interface-specific chapter (for example, "Using HP SICL with HP-IB") later in this manual.

Commander Sessions

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

Addressing Commander Sessions

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

hpib,cmdr	HP-IB commander session.
7,cmdr	Commander session on interface at logical unit 7.

The following are examples of creating a commander session with the HP-IB interface.

C example:

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

Visual BASIC example:

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

The above function calls will open a session of communication with the commander on the HP-IB interface.

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 both formatted I/O and non-formatted I/O routines:

- Formatted I/O converts mixed types of data under the control of a format string. The data is buffered, thus optimizing interface traffic. The formatted I/O routines are geared towards instruments, and reduce the amount of I/O code.
- Non-formatted I/O sends or receives raw data to a device, interface, or commander. With non-formatted I/O, no format or conversion of the data is performed. Thus, if formatted data is required, it must be done by the user.

See the following sections for a complete description and examples of using formatted I/O (for C applications and for Visual BASIC applications) and non-formatted I/O.

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 as follows:

- 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. 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. Refer to the "Formatted I/O Buffers" subsection later in this section for more 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 as follows:

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

See **iprintf**, **ipromptf**, and **iscanf** in the *HP SICL Reference Manual* for more information on how data is converted under the control of the format string.

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

Format Flags for `iprintf` and `ipromptf` in C Applications

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

The following example converts `numb` into a 488.2 floating point number and sends it to the session specified by *id*:

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

Sends: `61.000000`

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

The following example pads **numb** to six characters and sends it to the session specified by *id*:

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

Pads to six characters: `61`

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

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

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

Sends : `26.93`

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

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

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

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

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

Argument Modifiers
in C Applications

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

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

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

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

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

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

Sends 1024 characters of block data.

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

Conversion Character	Description
d,i,n	Corresponding argument must be a pointer to an integer.
e,f,g	Corresponding argument must be a pointer to a float.
c	Corresponding argument is a pointer to a character.
s,S,t	Corresponding argument is a pointer to a string.
o,u,x	Corresponding argument must be a pointer to an unsigned integer.
[	Corresponding argument must be a character pointer.
F	Corresponding argument is a pointer to a FILE descriptor opened for writing. (Not supported with `iscanf` on Windows 3.1.)

The following 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);
```

Formatted I/O C Example

The following C program example shows sending and receiving formatted I/O. This example opens an HP-IB communications session with a multimeter and uses a comma operator to send a comma separated list to the multimeter. The `lf` conversion characters are then used to receive a double from the multimeter.

```
/* formatio.c
   This example program makes a multimeter measurement with a comma
   separated list passed with formatted I/O and prints the results */

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

main()
{
  INST dvm;

  double res;
  double list[2] = {1,0.001};

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

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

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

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

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

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

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

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

/* For WIN16 programs, call _siclcleanup before exiting to release
   resources allocated by SICL for this application.  This call
   is a no-op for WIN32 programs. */
   _siclcleanup();

   return 0;
}
```

Format String

The format string for `iprintf` puts a special meaning on the newline character (`\n`). The newline character in the format string flushes the output buffer to the device. All characters in the output buffer will be written to the device with an END indicator included with the last byte (the newline character). This means that 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, then 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 the following subsection on "Formatted I/O Buffers."

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.

The write buffer is maintained by the `iprintf` and the write portion of the `ipromptf` functions. It queues characters to send to the device so that they are sent in large blocks, thus increasing performance. The write buffer automatically flushes when it sends a newline character from the format string (see the `%t` conversion character to change this feature). It 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. It queues the data received from a device until it is needed by the format string. The read buffer is automatically flushed before the write portion of an `ipromptf`. Flushing the read buffer destroys the data in the buffer and guarantees that the next call to `iscanf` or `ipromptf` reads data directly from the device rather than data that was previously queued.

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

See the `isetbuf` function for other options for buffering data.

Related Formatted I/O Functions

The following is a set of functions that are related to formatted I/O:

`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. Note that if no buffering is used, performance can be severely affected.
`isetubuf`	Sets the read or the write buffer to your allocated buffer. The same buffer cannot be used for both reading and writing. Also you should be careful when using buffers that are automatically allocated.

Formatted I/O in Visual BASIC Applications

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

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

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

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

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

NOTE

There are certain restrictions when using **ivprintf** *and* **ivscanf** *with Visual BASIC. For details about these restrictions, see either the "Restrictions Using* **ivprintf** *in Visual BASIC" section under the* **iprintf** *function, or the "Restrictions Using* **ivscanf** *in Visual BASIC" section under the* **iscanf** *function, in the HP SICL Reference Manual.*

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. Refer to the "Formatted I/O Buffers" subsection later in this section for more 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 as follows:

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

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

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

Format Flags for `ivprintf` in Visual BASIC Applications

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

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

```
Dim numb As Integer
Dim ret_val As Integer

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

Sends: `61.000000`

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

The following example pads **numb** to six characters and sends it to the session specified by *id*:

```
Dim numb As Integer
Dim ret_val As Integer

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

Pads to six characters: `61`

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

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

```
Dim numb As Double
Dim ret_val As Integer

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

Sends : **26.93**

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

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

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

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

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

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

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

Argument Modifiers in Visual BASIC Applications

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

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

`ivprintf` Conversion Characters in Visual BASIC Applications

Conversion Character	Description
d, i	Corresponding argument is an Integer.
b, B	Not supported on Visual BASIC.
c,C	Not supported on Visual BASIC.
t	Not supported on Visual BASIC.
s,S	Not supported on Visual BASIC.
%	Sends an ASCII percent (%) character.
o,u,x,X	Corresponding argument will be treated as an Integer.
f,e,E,g,G	Corresponding argument is a Double.
n	Corresponding argument is an Integer.
F	Corresponding *arg* is a pointer to a FILE descriptor. (Not supported with Visual BASIC on Windows 3.1.)

`ivscanf` Conversion Characters in Visual BASIC Applications

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

The following 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)
```

Formatted I/O Visual BASIC Example

The following Visual BASIC program example shows sending and receiving formatted I/O. This example opens an HP-IB communications session with a multimeter and uses a comma operator to send a comma separated list to the multimeter. The `lf` conversion characters are then used to receive a Double from the multimeter.

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

   On Error GoTo ErrorHandler

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

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

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

   ' Set up multimeter and send comma separated list
   nRetVal = ivprintf(dvm, "CALC:DBM:REF 50" + Chr$(10))
   nRetVal = ivprintf(dvm, "MEAS:VOLT:AC? %,21f" + Chr$(10), list())

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

   ' Display the results
   MsgBox "Result is " + Format$(res)

   ' Close the multimeter session
   Call iclose(dvm)

   ' Tell SICL to cleanup for this task
   Call siclcleanup
   End

ErrorHandler:
   ' Display the error message
   MsgBox "*** Error : " + Error$, MB_ICON_EXCLAMATION
   ' Tell SICL to cleanup for this task
   Call siclcleanup
   End
End Sub
```

Format String

In the format string for **ivprintf**, when the special characters **Chr$(10)** is used, the output buffer to the device is flushed. All characters in the output buffer will be written to the device with an END indicator included with the last byte. This means that you can control at what point you want the data written to the device. If no **Chr$(10)** is included in the format string for an **ivprintf** call, then 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. 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 **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

The following is a set of functions that are related to formatted I/O in Visual BASIC:

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

Non-Formatted I/O

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

The non-formatted I/O functions are described as follows:

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

  C example:

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

  Visual BASIC example:

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

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

  C example:

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

  Visual BASIC example:

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

Non-Formatted I/O Examples

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

C example:

```
/* nonfmt.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];
   unsigned long actual;

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

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

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

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

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

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

   /* NULL terminate result string and print the results */
   /* This technique assumes the last byte sent was a line-feed */
   if (actual) {
     strres[actual - 1] = (char) 0;
     printf("Result is %s\n", strres);
   }

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

```
    /* For WIN16 programs, call _siclcleanup before exiting to release
       resources allocated by SICL for this application.  This call
       is a no-op for WIN32 programs.  */
    _siclcleanup();

    return 0;
}
```

Visual BASIC example:

```
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'  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
   Dim actual As Long

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

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

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

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

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

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

'  Close the multimeter session
   Call iclose(dvm)

'  Tell SICL to cleanup for this task
   Call siclcleanup

   Exit Sub

End Sub
```

Handling Asynchronous Events in C Applications

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

NOTE

SICL allows you to install 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.

NOTE

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

NOTE

For WIN16 programs, custom SRQ and interrupt handler (callback) functions installed using SICL's `ionsrq` and `ionintr` functions should be declared using the SICL modifier `SICLCALLBACK`, which is defined as "`_export _far _pascal`" in `sicl.h`. Failure to do this usually causes a "General Protection Fault" error at the time the handler is called in 16-bit Windows.

Example declarations:

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

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

Additionally, if you are developing a 16-bit application using the QuickWin feature provided with Microsoft compilers and are installing a custom handler, you must also use the `_loadds` modifier with your handler declaration.

Example declaration for QuickWin applications:

```
void SICLCALLBACK _loadds 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. This disables 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.

NOTE

These functions do not affect the `isetintr` values or the handlers (`ionsrq` or `ionintr`) in any way. See `ionintr` and `ionsrq` in the *HP SICL Reference Manual*.

Default is **on**.

On operating systems that support multiple threads such as Windows 95 and Windows NT, SRQ and interrupt handlers execute on a separate thread (a thread created and managed by SICL). This means that 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 on's and off's. This means that 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 either an enabled SRQ or interrupt condition occurs and the related handler executes. Once the handler completes its operation, this function returns and processing continues.

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

NOTE

Interrupts must be disabled if you are using **iwaithdlr**. Use **iintroff** to disable interrupts.

The reason for disabling interrupts is because there may be a race condition between the **isetintr** and **iwaithdlr** and, if you only expect one interrupt, it might come before the **iwaithdlr**. This may or may not be the effect you desire.

For example:

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

Logging HP SICL Error Messages

Windows NT

SICL logs internal messages as Windows 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 may wish to view these messages. You can do so by starting the **Event Viewer** utility in the **Administrative Tools** group. Both system and application messages can be logged to the **Event Viewer** from SICL. SICL messages are identified either by `SICL LOG`, or by the driver name (for example, `hp341i32`).

Windows 95 and Windows 3.1

While developing your SICL application or tracking down problems in either Windows 95 or Windows 3.1, you may wish to 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 utility, double-click on the **Message Viewer** icon either in the `HP I/O Libraries` program group for Windows 95, or in the `HP SICL` program group for Windows 3.1. The utility must be started before execution of the SICL application. It will receive messages while minimized, however.

Using Error Handlers

Error handling is supported in C and Visual BASIC. Refer to the following subsection that applies to your programming language.

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 provides a convenient mechanism for handling errors. SICL allows you to install an error handler for all SICL functions within a C/C++ application.

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

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

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

Where:

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

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

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

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

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

NOTE

Custom error handler (callback) functions installed using SICL's `ionerror` function in WIN16 applications should be declared using the SICL modifier `SICLCALLBACK`, which is defined as "`_export _far _pascal`" in the `sicl.h` file. Failure to do this usually causes a "General Protection Fault" error at the time the handler is called in 16-bit Windows.

Example declarations:

```
void SICLCALLBACK my_err_handler(INST id, int error) {
   /* your code here */
}
```

Additionally, if you are developing a WIN16 application using the QuickWin feature provided with Microsoft compilers and are installing a custom handler, you must also use the `_loadds` modifier with your handler declaration.

Example declaration for QuickWin applications:

```
void SICLCALLBACK _loadds my_err_handler(INST id, int error) {
   /* your code here */
}
```

Error Handlers in C Examples

Typically in an application, error handling code is intermixed with the I/O code. 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 exits.

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

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

main ()
{
   INST dvm;
   double res;

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

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

  /* For WIN16 programs, call _siclcleanup before exiting to release
     resources allocated by SICL for this application.  This call
     is a no-op for WIN32 programs. */
     _siclcleanup();

   return 0;
}
```

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

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

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

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

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

   /* For WIN16 programs, call _siclcleanup before exiting to release
      resources allocated by SICL for this application.  This call
      is a no-op for WIN32 programs. */
      _siclcleanup();

   return 0;
}
```

NOTE

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

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 needs to 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 don't 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. Note that 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.

For more information on Visual BASIC error handlers, see the section "Handling Run-Time Errors" in the *Visual BASIC Programmer's Guide.*

Error Handlers in Visual BASIC Example

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

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

   On Error GoTo ErrorHandler

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

   ' Tell SICL to cleanup for this task
   Call siclcleanup
   End

ErrorHandler:
   ' Display the error message
   MsgBox "*** Error : " + Error$, MB_ICON_EXCLAMATION
   ' Tell SICL to cleanup for this task
   Call siclcleanup
   End
End Sub
```

Using Locks

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

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.

In particular, be aware that HP Visual Engineering Environment (HP VEE) applications use SICL interface sessions. Hence, I/O operations from these 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 simply 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 the section, "Using Locks and Multiple Threads over LAN," in Chapter 8, "Using HP SICL with LAN."

Lock Actions

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

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

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

Locking in a Multi-User Environment

In a multi-user/multi-process environment where devices are being shared, it is a good idea to use locking to ensure exclusive use of a particular device or set of devices. (However, as explained in the previous section, "Using Locks," remember that an interface session can access a device locked from a device session.) In general, it is not friendly behavior to lock a device at the beginning of an application and unlock it at the end. This can result in deadlock or long waits by others who want to use the resource.

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

Locking Examples

The following C and Visual BASIC examples show how device locking can be used to grant exclusive access to a device by an application.

C example:

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

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

main()
{
   INST dvm;

   char strres[20];
   unsigned long actual;

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

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

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

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

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

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

   /* Release the multimeter device for use by others */
   iunlock(dvm);
```

```
    /* NULL terminate result string and print 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);

    // For WIN16 programs, call _siclcleanup before exiting to release
    // resources allocated by SICL for this application.  This call
    // is a no-op for WIN32 programs.
    _siclcleanup();

    return 0;
}
```

Visual BASIC example:

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

'  Install an error handler
   On Error GoTo ErrorHandler

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

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

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

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

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

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

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

'  Tell SICL to cleanup for this task
   Call siclcleanup

   End

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

   End

End Sub
```

5

Using HP SICL with HP-IB

Using HP SICL with HP-IB

The HP-IB interface (Hewlett-Packard Interface Bus) is Hewlett-Packard Company's implementation of the IEEE 488.1 Bus. Other IEEE 488 versions include GPIB (General Purpose Interface Bus) and IEEE Bus. GPIB and HP-IB are both used synonymously in the discussions and examples in this chapter.

This chapter describes in detail how to open a communications session and communicate with HP-IB devices, interfaces, or controllers. The example programs shown in this chapter are also provided in the `C\SAMPLES\MISC` (for C/C++) and `VB\SAMPLES\MISC` (for Visual BASIC) subdirectories under the SICL base directory (for example, under `C:\SICL` if the default installation directory was used).

This chapter contains the following sections:

- Creating a Communications Session with HP-IB
- Communicating with HP-IB Devices
- Communicating with HP-IB Interfaces
- Communicating with HP-IB Commanders
- Writing HP-IB Interrupt Handlers
- Summary of HP-IB Specific Functions

NOTE

Using the HP 82335 HP-IB interface with both SICL and the HP-IB Command Library at the same time on the same interface is *not* supported. No error will be reported, but unexpected results could occur.

Creating a Communications Session with HP-IB

Once you have determined that your HP-IB system is setup and operating correctly, you may want to start programming with the SICL functions. First you must determine what type of communications session you need. The three types of communications sessions are device, interface, and commander.

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

Addressing HP-IB 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 **I/O Config** utility either from the **HP I/O Libraries** program group for Windows 95 or Windows NT, or from the **HP SICL** program group for Windows 3.1. See Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows* for information on the **I/O Config** utility.

The following are example HP-IB addresses for device sessions:

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

SICL supports both primary and secondary addressing on GPIB interfaces.

Remember that the primary address must be between 0 and 30 and that the secondary address must be between 0 and 30. The primary and secondary addresses correspond to the HP-IB primary and secondary addresses.

NOTE

If you are connecting to a VXI cardcage through an HP E1405/06 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 cardcage. Secondary addresses of 0, 1, 2, ... 30 correspond to VXI instruments at logical addresses of 0, 8, 16, ... 240, respectively. See "HP-IB Device Session Examples" later in this chapter for an example program of communicating with a VXI cardcage over the HP-IB interface.

The following are examples of opening a device session with an HP-IB device at bus address 16.

C example:

```
INST dmm;
dmm = iopen ("hpib,16");
```

Visual BASIC example:

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

HP SICL Function Support with HP-IB Device Sessions

The following describes how some SICL functions are implemented for HP-IB device sessions. The data transfer functions work only when the HP-IB interface is the Active Controller. Passing control to another HP-IB device causes this device to lose active control.

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

HP-IB Device Session Interrupts

There are no device-specific interrupts for the HP-IB interface.

HP-IB Device Sessions and Service Requests

HP-IB device sessions support Service Requests (SRQs). On the HP-IB interface, when one device issues an SRQ, the library will inform *all* HP-IB device sessions that have SRQ handlers installed (see `ionsrq` in the *HP SICL Reference Manual*). This is an artifact of how HP-IB handles the SRQ line. The interface cannot distinguish which device requested service; therefore, the library acts as if all devices require service. Your SRQ handler can retrieve the device's status byte by using the `ireadstb` function. For more information, see the section "Writing HP-IB Interrupt Handlers" later in this chapter.

HP-IB Device Session Examples

The following examples illustrate communicating with an HP-IB device session. These examples open two HP-IB communications sessions with VXI devices (through 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.

C example:

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

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

main()
{
  INST dvm;
  INST sw;

  double res;
  int i;

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

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

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

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

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

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

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

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

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

    /* For WIN16 programs, call _siclcleanup before exiting to release
       resources allocated by SICL for this application.  This call
       is a no-op for WIN32 programs. */
       _siclcleanup();

       return 0;
}
```

Visual BASIC example:

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

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

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

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

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

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

   ' Display form1 and print voltage measurements
   form1.Show

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

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

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

      ' Trigger to close channel
      argcount = ivprintf(sw, "TRIG" + Chr$(10))
   Next i

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

```
    ' Close the switch session
    Call iclose(sw)

    ' Tell SICL to cleanup for this task
    Call siclcleanup

End Sub
```

Communicating with HP-IB Interfaces

Interface sessions allow you direct, low-level control of the specified interface. You must do all the bus maintenance for the interface. This also implies that you know a lot about the interface. Additionally, when using interface sessions, you need to use interface-specific functions. The use of these functions means that the program can not be used on other interfaces and, therefore, becomes less portable.

Addressing HP-IB Interfaces

To create an interface session on your HP-IB 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 **I/O Config** utility either from the **HP I/O Libraries** program group for Windows 95 or Windows NT, or from the **HP SICL** program group for Windows 3.1. See Chapter 2, "Installing and Configuring the HP I/O Libaries," in the *HP I/O Libraries Installation and Configuration Guide for Windows* for information on the **I/O Config** utility.

The following are example interface addresses:

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

The following are examples that open an interface session with the HP-IB interface.

C example:

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

Visual BASIC example:

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

HP SICL Function Support with HP-IB Interface Sessions

The following describes how some SICL functions are implemented for HP-IB interface sessions.

`iwrite` Sends the specified bytes directly to the interface without performing any bus addressing. The **`iwrite`** function always clears the ATN line before sending any bytes, thus ensuring that the GPIB interface sends the bytes as data, not 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.

HP-IB Interface Session Interrupts

There are specific interface session interrupts that can be used. See **`isetintr`** in the *HP SICL Reference Manual* for information on the interface session interrupts for HP-IB. Also see the section "Writing HP-IB Interrupt Handlers" later in this chapter for more information.

HP-IB Interface Sessions and Service Requests

HP-IB interface sessions support Service Requests (SRQs). On the HP-IB interface, when one device issues an SRQ, the library will inform *all* HP-IB interface sessions that have SRQ handlers installed (see `ionsrq` in the *HP SICL Reference Manual*). For more information, see the section "Writing HP-IB Interrupt Handlers" later in this chapter.

HP-IB Interface Session Examples

The following example programs retrieve the HP-IB interface bus status information and displays it for the user.

C example:

```
/* hpibstat.c
   The following example retrieves and displays
   HP-IB 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 HP-IB interface session */
   id = iopen("hpib");
   itimeout (id, 10000);
```

```
    /* retrieve HP-IB 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);

    /* For WIN16 programs, call _siclcleanup before exiting to release
       resources allocated by SICL for this application.  This call
       is a no-op for WIN32 programs. */
       _siclcleanup();

    return 0;
}
```

Visual BASIC example:

```
'hpibstat.bas
'  The following example retrieves and displays
'  HP-IB 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 HP-IB interface session
   id = iopen("hpib7")
   Call itimeout(id, 10000)

   ' Retrieve HP-IB 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 cleanup for this task
   Call siclcleanup

End Sub
```

Communicating with HP-IB Commanders

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

Addressing HP-IB Commanders

To create a commander session on your HP-IB 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 **I/O Config** utility either from the **HP I/O Libraries** program group for Windows 95 or Windows NT, or from the **HP SICL** program group for Windows 3.1. See Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows* for information on the **I/O Config** utility.

The following are example HP-IB addresses for commander sessions:

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

The following are examples that open a commander session with the HP-IB interface.

C example:

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

Visual BASIC example:

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

HP SICL Function Support with HP-IB Commander Sessions

The following describes how some SICL functions are implemented for HP-IB commander sessions.

`iwrite` If the interface has been addressed to talk, the data is written directly to the interface. If the interface has not been addressed to talk, it will wait to be addressed to talk before writing the data.

`iread` If the interface has been addressed to listen, the data is read directly from the interface. If the interface has not been addressed to listen, it will wait to be addressed to listen before reading the data.

`isetstb` Sets the status value that will be returned on a **`ireadstb`** call (that is, when this device is SPOLLed). Bit 6 of the status byte has a special meaning. If bit 6 is set, the SRQ line will be set. If bit 6 is clear, the SRQ line will be cleared.

HP-IB Commander Session Interrupts

There are specific commander session interrupts that can be used. See **`isetintr`** in the *HP SICL Reference Manual* for information on the commander session interrupts. Also see the following section, "Writing HP-IB Interrupt Handlers," for more information.

Writing HP-IB Interrupt Handlers

This section provides some additional information you should be aware of when writing interrupt handlers for HP-IB 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 HP-IB application should be written to handle this situation gracefully. This can be done by keeping track of the current talk/listen state of the interface card, and ignoring the interrupt if the state does not change. For more information, see the *secval* parameter definition of the **isetintr** function in the *HP SICL Reference Manual*.

Handling SRQs from Multiple HP-IB Instruments

HP-IB is a multiple-device bus, and SICL allows multiple device sessions open at the same time. On the HP-IB interface, when one device issues a Service Request (SRQ), the library will inform *all* HP-IB device sessions that have SRQ handlers installed (see **ionsrq** in the *HP SICL Reference Manual*). This is an artifact of how HP-IB handles the SRQ line; the underlying HP-IB hardware does not support session-specific interrupts like VXI does. Therefore, your application must reflect the nature of the HP-IB hardware if you expect to reliably service SRQs from multiple devices on the same HP-IB 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 is:

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

This algorithm loops through all the device sessions and does not exit until the SRQ line is released (not asserted). The following example shows a SICL program segment which 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.

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

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

/* Service all sessions in turn until no one is
   requesting service */
while (srq_asserted) {
   ireadstb(id1, &statusbyte);
   if (statusbyte & SRQ_BIT) {
     /* Actual service actions depend upon application */
     iscanf(id1, "%f", &data1);
   }
   ireadstb(id2, &statusbyte);
   if (statusbyte & SRQ_BIT) {
     iscanf(id2, "%f", &data2);
   }
   igpibbusstatus(bus, I_GPIB_BUS_SRQ, &srq_asserted);
  }
}

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

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

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

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 simply to perform the **`ireadstb`** and set a flag in the handler. Then the main program can test the flags for each device and perform the more lengthy service.

Even if the different device sessions are in different processes, it is still important to stay in the SRQ handler until the SRQ line is released. However, it is not likely that a process which only knows about Device A can do anything to make Device B release the SRQ line. In such a configuration, a single unserviced instrument can effectively disable SRQs for all processes attempting to use that interface. Again, this is a hardware characteristic of HP-IB. The only way to ensure true independence of multiple HP-IB processes is to use multiple HP-IB interfaces.

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
igpiblllo	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

6

Using HP SICL with GPIO

Using HP SICL with GPIO

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

NOTE

GPIO is *only* supported with SICL on Windows 95 and Windows NT.

GPIO is *not* supported with SICL over LAN.

This chapter describes in detail how to open a communications session and communicate with an instrument over a GPIO connection. The example programs shown in this chapter are also provided in the `C\SAMPLES\MISC` (for C/C++) and `VB\SAMPLES\MISC` (for Visual BASIC) subdirectories under the SICL base directory (for example, under `C:\SICL` if the default installation directory was used).

This chapter contains the following sections:

- Creating a Communications Session with GPIO
- Communicating with GPIO Interfaces
- Summary of GPIO Specific Functions

Creating a Communications Session with GPIO

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

With HP-IB, there can be multiple devices on a single interface. These interfaces support a connection called a **device session**. With GPIO, only one device is connected to the interface. Therefore, you communicate with GPIO devices using an **interface session**.

Communicating with GPIO Interfaces

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

Addressing GPIO Interfaces

To create an interface session on GPIO, specify the interface logical unit or symbolic name in the *addr* parameter of the **iopen** function. The interface logical unit and symbolic name are defined by running the **I/O Config** utility from the **HP I/O Libraries** program group. See Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows* for information on the **I/O Config** utility.

The following are example addresses for GPIO interface sessions:

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

The following example opens an interface session with the GPIO interface:

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

HP SICL Function Support with GPIO Interface Sessions

The following describes how some SICL functions are implemented for GPIO interface sessions.

`iwrite`, `iread` — The *size* parameters for non-formatted I/O functions are always byte counts, regardless of the current data width of the interface.

`iprintf`, `iscanf` — All formatted I/O functions work with GPIO. When formatted I/O is used with 16-bit data widths, the formatting buffers re-assemble the data as a stream of bytes. On Windows 95, these bytes are ordered: high-low-high-low . . . Because of this "unpacking" operation, 16-bit data widths may not be appropriate for formatted I/O operations. For **`iscanf`** termination, an END value must be specified using **`igpioctrl`**. See the *HP SICL Reference Manual* for details.

`itermchr` — With 16-bit data widths, only the low (least-significant) byte is used.

`ixtrig` — Provides a method of triggering using either the CTL0 or CTL1 control lines. This function pulses the specified control line for approximately 1 or 2 microseconds. The following constants are defined:

`I_TRIG_STD`	Pulse CTL0 line
`I_TRIG_GPIO_CTL0`	Pulse CTL0 line
`I_TRIG_GPIO_CTL1`	Pulse CTL1 line

`itrigger` — Same as **`ixtrig`** (`I_TRIG_STD`). Pulses the CTL0 control line.

`iclear` — Pulses the P_RESET line for at least 12 microseconds, aborts any pending writes, discards any data in the receive buffer, and resets any error conditions. Optionally clears the Data Out port, depending upon the configuration specified via the **`I/O Config`** utility.

`ionsrq`	Installs a service request handler for this session. The concept of service request (SRQ) originates from HP-IB. On an HP-IB interface, a device can request service from the controller by asserting a line on the interface bus. On GPIO, the EIR line is assumed to be the service request line.
`ireadstb`	The *HP SICL Reference Manual* says that `ireadstb` is for device sessions only. Since GPIO has no device sessions, `ireadstb` is allowed with GPIO interface sessions. The interface status byte has bit 6 set if EIR is asserted; otherwise, the status byte is 0 (zero). This allows normal SRQ programming techniques in GPIO SRQ handlers.

GPIO Interface Session Interrupts

There are specific interface session interrupts that can be used. See `isetintr` in the *HP SICL Reference Manual* for information on the interface session interrupts for GPIO.

GPIO Interface Session Examples

C example:

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

#include <sicl.h>

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

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

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

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

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

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

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

   /* get the data */
   iscanf(id, "%f%*t", &result);
   printf("Result = %f\n", result);
```

```
    /* For Windows 3.1, call _siclcleanup before exiting to release
       resources allocated by SICL for this application.  This call
       is a no-op for WIN32 applications.
    */
    _siclcleanup();

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

Visual BASIC example:

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

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

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

'  Open an interface session using a known symbolic name
   id = iopen("gpio12")

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

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

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

'  Get the data
   retval = ivscanf(id, "%lf%*t", real_data)

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

'  Close the device session.
   Call iclose(id)

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

   Exit Sub
```

```
ErrorHandler:

'  Display the error message string in a Message Box
   retval = MsgBox(Error$, MB_ICONEXCLAMATION, "SICL Error")

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

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

   Exit Sub

End Sub

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'  The following routine is called when the application's
'  Start Up form is unloaded.  It calls siclcleanup to
'  release resources allocated by SICL for this
'  application.
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Sub Form_Unload (Cancel As Integer)
   Call siclcleanup     ' Tell SICL to clean up for this task
End Sub
```

GPIO Interrupts Example

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

#include <sicl.h>

void SICLCALLBACK handler(id, reason, sec)
INST id;
int reason, sec;
{
   if (reason == I_INTR_GPIO_EIR) {
      printf("EIR interrupt detected\n");

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

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

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

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

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

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

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

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

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

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

    /* For Windows 3.1, call _siclcleanup before exiting to release
       resources allocated by SICL for this application.  This call
       is a no-op for WIN32 applications.
    */
    _siclcleanup();
}
```

Summary of GPIO Specific Functions

Function Name — Action

`igpioctrl` — Sets the following characteristics of the GPIO interface:

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

`igpiogetwidth` — Returns the current width (in bits) of the GPIO data ports.

`igpiosetwidth` — Sets the width (in bits) of the GPIO data ports. Either 8 or 16.

Function Name	Action
`igpiostat`	Gets the following information about the GPIO interface:

Request	Characteristic	Value
`I_GPIO_CTRL`	Control Lines	`I_GPIO_CTRL_CTL0` `I_GPIO_CTRL_CTL1`
`I_GPIO_DATA`	Data In lines	16-bit mask

`I_GPIO_INFO`	GPIO information	`I_GPIO_AUTO_HDSK`
		`I_GPIO_CHK_PSTS`
		`I_GPIO_EIR`
		`I_GPIO_ENH_MODE`
		`I_GPIO_PSTS`
		`I_GPIO_READY`

`I_GPIO_READ_EOI`	END termination pattern	`I_GPIO_EOI_NONE` or 8-bit or 16-bit mask
`I_GPIO_STAT`	Status lines	`I_GPIO_STAT_STI0` `I_GPIO_STAT_STI1`

7

Using HP SICL with RS-232

Using HP SICL with RS-232

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

This chapter describes in detail how to open a communications session and communicate with an instrument over an RS-232 connection. The example programs shown in this chapter are also provided in the `C\SAMPLES\MISC` (for C/C++) and `VB\SAMPLES\MISC` (for Visual BASIC) subdirectories under the SICL base directory (for example, `C:\SICL` if the default installation directory was used).

This chapter contains the following sections:

- Creating a Communications Session with RS-232
- Communicating with RS-232 Devices
- Communicating with RS-232 Interfaces
- Summary of RS-232 Specific Functions

Creating a Communications Session with RS-232

Once you have configured your system for RS-232 communications, you can start programming with the SICL functions. If you have programmed RS-232 before, you will probably want to open the interface and start sending commands. With SICL, you must first determine what type of communications session you will need.

SICL is designed to provide a standard way of accessing instrumentation that is independent from the type of connection. With HP-IB, there can be multiple devices on a single interface. SICL allows you direct access to a device on an interface without worrying about the type of interface to which it is connected. To do this, you communicate with a device session. SICL also allows you to do interface-specific actions, such as setting up device addresses or setting other interface-specific characteristics. To do this, you communicate with an interface session.

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

Using SICL to communicate with an instrument on RS-232 is similar to using SICL over HP-IB. You must first determine what type of communications session you will need. An RS-232 communications session can be either a device session or an interface session. Commander sessions are not supported on RS-232.

An RS-232 device session should be used when sending commands and receiving data from an instrument. Setting interface characteristics (such as the baud rate) must be done with an interface session.

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

Addressing RS-232 Devices

To create a device session, specify the interface logical unit or symbolic name, followed by a device logical address of **488**. The interface logical unit and symbolic name are defined by running the **I/O Config** utility either from the **HP I/O Libraries** program group for Windows 95 or Windows NT, or from the **HP SICL** program group for Windows 3.1. See Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows* for information on running **I/O Config**. The device address of **488** tells SICL that you are communicating with an instrument that uses the IEEE 488.2 standard command structure.

NOTE

If your instrument does not "speak" IEEE 488.2, you can still use SICL to communicate with it. However, some of the SICL functions that work only with device sessions may not operate correctly. See the next section, "HP SICL Function Support with RS-232 Device Sessions."

The following are example addresses for RS-232 device sessions:

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

For other interfaces, SICL supports the concept of primary and secondary addresses. For RS-232, the only primary address supported is `488`. SICL does not support secondary addressing on RS-232 interfaces.

The following are 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")
```

HP SICL Function Support with RS-232 Device Sessions

The following describes how some SICL functions are implemented for RS-232 device sessions.

`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 the section titled "HP SICL Function Support with RS-232 Interface Sessions" later in this chapter.
`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. Note that 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`**). Note that 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 the following function:

```
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 the section titled "HP SICL Function Support for RS-232 Interface Sessions" later in this chapter.

RS-232 Device Session Interrupts

There are specific device session interrupts that can be used. See **`isetintr`** in the *HP SICL Reference Manual* for information on the device session interrupts for RS-232.

RS-232 Device Session Examples

NOTE

The following **ser_dev** example programs were tested with an HP 34401A Digital Voltmeter. When you run the program with a serial connection to the HP 34401A, make sure that DTR/DSR flow control is set for the serial port. Otherwise, the program will appear not to work.

C example:

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

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

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

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

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

main()
{
   INST dvm;
   double res;

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

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

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

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

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

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

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

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

    // For WIN16 programs, call _siclcleanup before exiting to release
    // resources allocated by SICL for this application.  This call
    // is a no-op for WIN32 programs.
    _siclcleanup();

    return 0;
}
```

Visual BASIC example:

```
'  ser_dev.bas
'  This example program takes a measurement from a DVM using a
'  SICL device session.
Sub Main ()
   Dim dvm As Integer
   Dim res As Double
   Dim argcount As Integer

   ' Open the multimeter session
   dvm = iopen("COM1,488")
   Call itimeout(dvm, 10000)

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

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

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

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

   ' Print the results
   MsgBox "Result is " + Format(res), MB_ICON_EXCLAMATION

   ' Close the multimeter session
   Call iclose(dvm)

'  Tell SICL to cleanup for this task
   Call siclcleanup

End Sub
```

Interface sessions can be used to get or set the characteristics of the RS-232 connection. Examples of some of these characteristics are baud rate, parity, and flow control.

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 **I/O Config** utility either from the **HP I/O Libraries** program group for Windows 95 or Windows NT, or from the **HP SICL** program group for Windows 3.1. See Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows* for information on running **I/O Config**.

The following are example addresses for RS-232 interface sessions:

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

The following 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")
```

HP SICL Function Support with RS-232 Interface Sessions

The following describes how some SICL functions are implemented for RS-232 interface sessions.

`iwrite`, `iread` All I/O functions (non-formatted and formatted) work the same as for device sessions. However, it is recommended that all I/O be performed with device sessions to make your programs easier to maintain.

`ixtrig` Provides a method of triggering using either the DTR or RTS modem status line. This function clears the specified modem status line, waits 10 milliseconds, then sets it again. Specifying `I_TRIG_STD` is the same as specifying `I_TRIG_SERIAL_DTR`.

`itrigger` Pulses the DTR modem control line for 10 milliseconds.

`iclear` Sends a break, aborts any pending writes, discards any data in the receive buffer, resets any flow control states (such as **`XON/XOFF`**), and resets any error conditions. To reset the interface without sending a break, use the following function:

```
iserialctrl (id, I_SERIAL_RESET, 0)
```

`ionsrq` Installs a service request handler for this session. The concept of service request (SRQ) originates from HP-IB. On an HP-IB 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 `I/O Config` utility. Any transition on the designated service request line will cause an SRQ handler in your program to be called. (Be sure not to set the SRQ line to CTS or DSR if you are also using that line for hardware flow control.)

Service requests are supported for both device sessions and interface sessions. When the designated SRQ line changes state, the RS-232 driver calls all SRQ handlers installed by either device sessions or interface sessions.

`iserialctrl` Sets the characteristics of the serial interface. The following requests are clarified:

- `I_SERIAL_DUPLEX`: The duplex setting determines whether data can be sent and received simultaneously. Setting full duplex allows simultaneous send and receive data traffic. Setting half duplex (the default) will cause reads and writes to be interleaved, so that data is flowing in only one direction at any given time. (The exception to this is if `XON/XOFF` flow control is used.)
- `I_SERIAL_READ_BUFSZ`: The default read buffer size is 2048 bytes.
- `I_SERIAL_RESET`: Performs the same function as the `iclear` function on an interface session, except that a break is not sent.

`iserialstat` Gets the characteristics of the serial interface. The following requests are clarified:

- `I_SERIAL_MSL`: Gets the state of the modem status line. Because of the way Windows supports RS-232, the `I_SERIAL_RI` bit will never be set. However, the `I_SERIAL_TERI` bit will be set when the RI modem status line changes from high to low.
- `I_SERIAL_STAT`: Gets the status of the transmit and receive buffers and the errors that have occurred since the last time this request was made. Only the error bits (`I_SERIAL_PARITY`, `I_SERIAL_OVERFLOW`, `I_SERIAL_FRAMING`, and `I_SERIAL_BREAK`) are cleared; the `I_SERIAL_READ_ DAV` and `I_SERIAL_TEMT` bits reflect the status of the buffers at all times.
- `I_SERIAL_READ_DAV`: Gets the current amount of data available for reading. This shows how much data is in Windows' receive buffer, not how much data is in the buffer used by the formatted input functions such as `iscanf`.

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

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

RS-232 Interface Session Interrupts

There are specific interface session interrupts that can be used. See `isetintr` in the *HP SICL Reference Manual* for information on the interface session interrupts for RS-232.

RS-232 Interface Session Examples

C example:

```
/*ser_intf.c
   This program does the following:
   1) gets the current configuration of the serial port,
   2) sets it to 9600 baud, no parity, 8 data bits, and
      1 stop bit, and
   3) 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();   // required for Borland EasyWin programs
   #endif

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

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

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

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

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

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

    /* close port */
    iclose (intf);

    // For WIN16 programs, call _siclcleanup before exiting to release
    // resources allocated by SICL for this application.  This call
    // is a no-op for WIN32 programs.
    _siclcleanup();

    return 0;
}
```

Visual BASIC example:

```
' ser_intf.bas
' This program does the following:
' 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
   intf = iopen("COM1")
   Call itimeout(intf, 10000)

   ' get baud rate, parity, data bits, and stop bits
   Call iserialstat(intf, I_SERIAL_BAUD, baudrate)
   Call iserialstat(intf, I_SERIAL_PARITY, parity)
   Call iserialstat(intf, I_SERIAL_WIDTH, databits)
   Call iserialstat(intf, I_SERIAL_STOP, stopbits)

   ' determine string to display for parity
   Select Case parity
   Case I_SERIAL_PAR_NONE
      parity_str = "NONE"
   Case I_SERIAL_PAR_ODD
      parity_str = "ODD"
   Case I_SERIAL_PAR_EVEN
      parity_str = "EVEN"
   Case I_SERIAL_PAR_MARK
      parity_str = "MARK"
   Case Else
      parity_str = "SPACE"
   End Select

   ' set to 9600,NONE,8, 1
   Call iserialctrl(intf, I_SERIAL_BAUD, 9600)
   Call iserialctrl(intf, I_SERIAL_PARITY, I_SERIAL_PAR_NONE)
   Call iserialctrl(intf, I_SERIAL_WIDTH, I_SERIAL_CHAR_8)
   Call iserialctrl(intf, I_SERIAL_STOP, I_SERIAL_STOP_1)
```

```
    ' display previous settings
    msg_str = "Old settings: " + Str$(baudrate) + "," + parity_str + ","
+ Str$(databits) + "," + Str$(stopbits)
    MsgBox msg_str, MB_ICON_EXCLAMATION

    ' close port
    Call iclose(intf)

'   Tell SICL to cleanup for this task
    Call siclcleanup

End Sub
```

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

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

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

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

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

8

Using HP SICL with LAN

Using HP SICL with LAN

This chapter explains how to use SICL over LAN (Local Area Network). LAN is a natural way to extend the control of instrumentation beyond the limits of typical instrument interfaces.

NOTE

LAN is *only* supported with 32-bit SICL on Windows 95 and Windows NT. If you are programming in Visual BASIC, this means that LAN is *only* supported with 32-bit Visual BASIC version 4.0.

Also, the GPIO interface is *not* supported with SICL over LAN.

This chapter describes in detail how to open a communications session and communicate with devices over LAN. The example programs shown in this chapter are also provided in the `C\SAMPLES\MISC` (for C/C++) and `VB\SAMPLES\MISC` (for Visual BASIC) subdirectories under the SICL base directory (for example, under `C:\SICL` if the default installation directory was used).

This chapter contains the following sections:

- Overview of LAN with HP SICL
- Considering LAN Configuration and Performance
- Communicating with LAN Devices
- Using Locks and Multiple Threads over LAN
- Using Timeouts with LAN
- Summary of LAN Specific Functions

NOTE

To start the LAN server on a Windows 95 or Windows NT system, see the appropriate "Starting the LAN Server" section of Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows*.

To stop the LAN server on a Windows 95 or Windows NT system, see the appropriate "Stopping the LAN Server" section of Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows*.

Overview of LAN with HP SICL

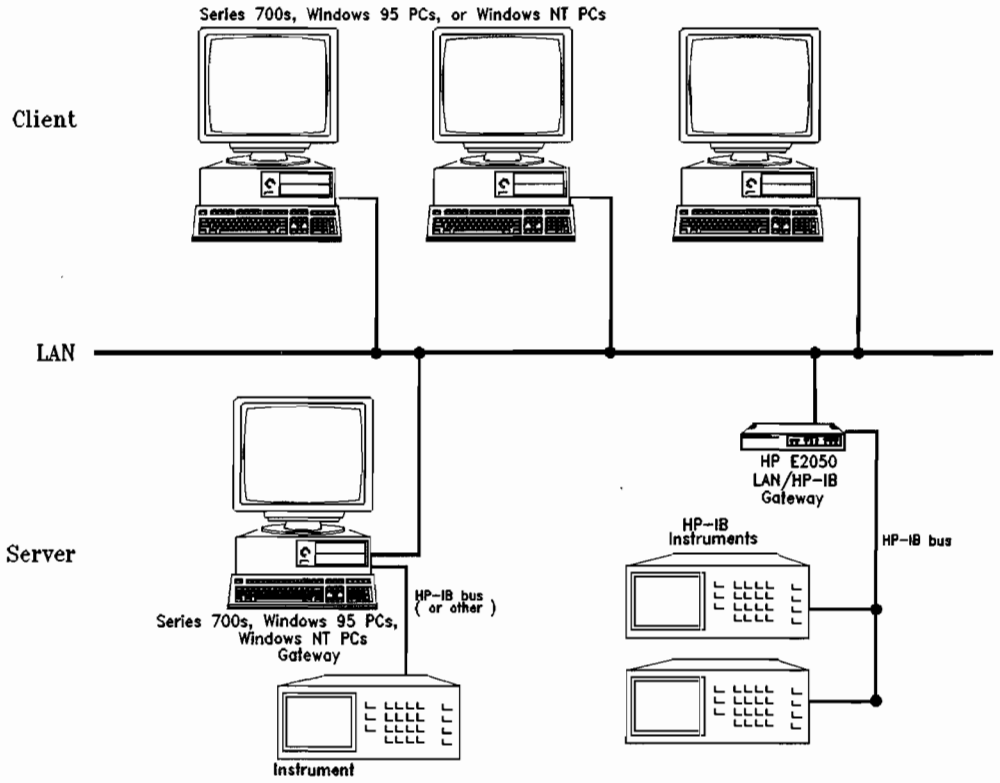
The LAN software provided with SICL allows you to control instrumentation over a LAN. LAN connections are included on many systems being sold today. By making use of these standard LAN connections, instrument control can be driven from a computer which does not have a special interface for instrument control.

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:

- Resource sharing by multiple applications/people within an organization.
- 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 themselves.

As shown in the following figure, a LAN client computer system (a Series 700 HP-UX workstation, a Windows 95 PC, or a Windows NT PC) makes SICL requests over the network to a LAN server (a Series 700 HP-UX workstation, a Windows 95 PC, a Windows NT PC, or an HP E2050 LAN/HP-IB 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 which indicates whether the operation was successful.

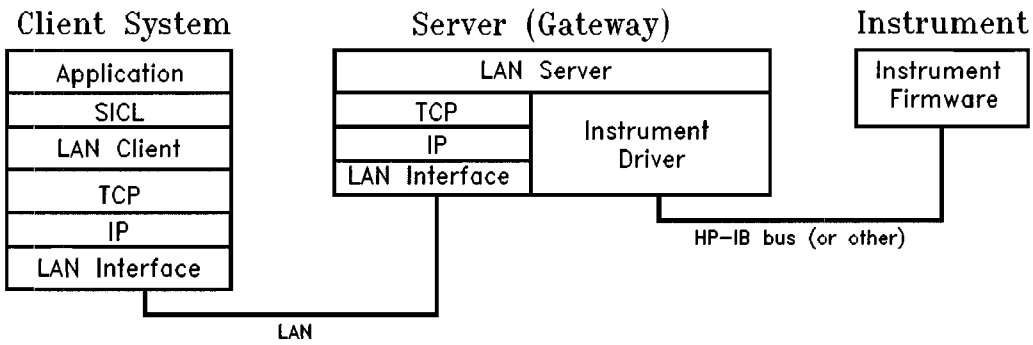


Using the LAN Client and LAN Server (Gateway)

The LAN server acts as a gateway between the LAN that your client system supports, and the instrument-specific interface that your device supports. Due to the LAN server's gateway functionality, we refer to devices or interfaces which 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 the following figure shows, 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 it.



LAN Software Architecture

LAN Networking Protocols

The LAN software provided with SICL is built on top of standard LAN networking protocols. There are two LAN networking protocols provided with the SICL software. You can choose one or both of these protocols when configuring your systems (via the `I/O Config` utility) to use SICL over LAN. The two protocols are as follows:

- SICL LAN Protocol is a networking protocol developed by HP which is compatible with all existing SICL LAN products. This LAN networking protocol is the default choice in the `I/O Config` utility when you are configuring LAN for SICL. The SICL LAN Protocol on Windows 95 and Windows NT supports SICL operations over the LAN to HP-IB/GPIB and RS-232 interfaces.
- TCP/IP Instrument Protocol is a networking protocol developed by the VXIbus Consortium based on the SICL LAN Protocol which permits interoperability of LAN software from different vendors that meet the VXIbus Consortium standards. Note that this LAN networking protocol may not be implemented with all the SICL LAN products at this time. The TCP/IP Instrument Protocol on Windows 95 and Windows NT supports SICL operations over the LAN to HP-IB/GPIB interfaces. Also, some SICL operations are not supported when using the TCP/IP Instrument Protocol. See the section titled "HP SICL Function Support with LAN-gatewayed Sessions" later in this chapter.

When using either of these networking protocols, the LAN software provided with SICL uses the TCP/IP protocol suite to pass messages between the LAN client and the LAN server. The server accepts device I/O requests over the network from the client and then proceeds to execute those I/O requests on a local interface, such as HP-IB.

You can use both LAN networking protocols with a LAN client. To do so, simply configure *both* the SICL LAN Protocol and the TCP/IP Instrument Protocol on the LAN client system via the `I/O Config` utility. (See Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows* for information on running `I/O Config`.) Then use the name of the interface supporting the protocol you wish to use in each SICL `iopen` call of your program. (See the "Communicating with LAN Devices" section later in this chapter for details on how to create communications sessions with SICL over LAN using each of these protocols.) Note, however, that the LAN server does *not* support simultaneous connections from LAN clients using the SICL LAN Protocol and from other LAN clients using the TCP/IP Instrument Protocol.

LAN Client 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. Note that requests are handled sequentially even if they are intended for different LAN servers.

If you want concurrent threads to be processed simultaneously with SICL over LAN, use multiple processes. For more information on using threads in WIN32 SICL applications, refer to the section, "Thread Support for 32-bit Windows Applications," in Chapter 3, "Building an HP SICL Application." Also see the section, "Using Locks and Multiple Threads over LAN," later in this chapter for information on using locks in multi-threaded applications.

LAN Server

SICL includes the necessary software to allow a Windows 95 PC or a Windows NT PC to act as a LAN-to-instrument_interface gateway. To use this capability, the PC must have a local interface configured for I/O. The supported interfaces for this release are GPIB/HP-IB and RS-232 with the SICL LAN Protocol, and GPIB/HP-IB with the TCP/IP Instrument Protocol. (The LAN server does *not* support VXI operations with either protocol.)

Note that the timing of operations performed remotely over a network will be different from the 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.

Contact your local HP representative for a current list of other HP supported LAN servers.

Considering LAN Configuration and Performance

As with other client/server applications on a LAN, when deploying an application which 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 utilization of the LAN must be considered. Depending on the amount of data which will 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 simply 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.

Communicating with LAN Devices

There are several different types of sessions which are supported over LAN. This section describes those session types and what behavior should be expected for the various SICL calls.

LAN-gatewayed Sessions

Communicating with a device over LAN through a LAN-to-instrument_interface gateway preserves the functionality of the gatewayed-interface with only a few exceptions. (See the "HP SICL Function Support with LAN-gatewayed Sessions" section later in this chapter.) This means most operations you might request of an interface, such as HP-IB, connected directly to your controller, you can also request of a remote interface via the LAN gateway. The only portions of your application which must change 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 to it so that the SICL software knows to direct the request to a LAN server on the network.

Addressing Devices or Interfaces with LAN-gatewayed Sessions

To create a LAN-gatewayed session, specify the LAN's interface logical unit or interface name, the IP address or hostname of the server machine, and the address of the remote interface or device in the *addr* parameter of the **iopen** function. The interface logical unit and interface name are defined by running the **I/O Config** utility from the **HP I/O Libraries** program group. See Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows* for information on running **I/O Config**.

The following are examples of LAN-gatewayed addresses:

`lan[instserv]:GPIB,7`	A device address corresponding to the device at primary address 7 on the `GPIB` interface attached to the machine named **`instserv`**.
`lan[instserv.hp.com]:hpib,7`	A device address corresponding to the device at primary address 7 on the **`hpib`** interface attached to the machine named **`instserv`** in the **`hp.com`** domain. (Fully qualified domain names may be used.)
`lan1[128.10.0.3]:GPIB0,3,2`	A device address corresponding to the device at primary address 3, secondary address 2, on the `GPIB0` interface attached to the machine with IP address `128.10.0.3`.
`lan1[intserv]:GPIB2`	An interface address corresponding to the `GPIB2` interface attached to the machine named **`intserv`**.
`30,intserv:hpib,3,2`	A device address corresponding to the device at primary address 3, secondary address 2, on the **`hpib`** interface attached to the machine named **`intserv`**. (30 is the default logical unit for LAN.)
`lan[intserv]:GPIB,cmdr`	A commander session with the `GPIB` interface attached to the machine named **`intserv`**. (This example assumes that the server supports GPIB commander sessions).

NOTE

If you are using the IP address of the server machine rather than the hostname, then you cannot use the comma notation, but must use the bracket notation.

Incorrect:

```
lan,128.10.0.3:hpib
```

Correct:

```
lan[128.10.0.3]:hpib
```

The following table shows the relationship between the address passed to **iopen**, the session type returned by **igetsesstype**, the interface type returned by **igetintftype**, and the value returned by **igetgatewaytype**.

Address	Session Type	Interface Type	Gateway Type
lan	I_SESS_INTF	I_INTF_LAN	I_INTF_NONE
lan[instserv]:hpib	I_SESS_INTF	I_INTF_GPIB	I_INTF_LAN
lan[instserv]:hpib,7	I_SESS_DEV	I_INTF_GPIB	I_INTF_LAN
hpib	I_SESS_INTF	I_INTF_GPIB	I_INTF_NONE
hpib,7	I_SESS_DEV	I_INTF_GPIB	I_INTF_NONE

HP SICL Function Support with LAN-gatewayed Sessions

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

The following SICL functions are *not* supported over LAN using either protocol:

- `iblockcopy`
- `imap`
- `imapinfo`
- `ipeek`
- `ipoke`
- `ipopfifo`
- `ipushfifo`
- `iunmap`

The following SICL functions, in addition to those listed above, are *not* supported with the TCP/IP Instrument Protocol:

- All RS-232/serial specific functions
- `igetlu`
- `ionintr`
- `isetintr`
- `igetintfsess`
- `igetonintr`
- `igpibgett1delay`
- `igpibppoll`
- `igpibppollconfig`
- `igpibppollresp`
- `igpibsett1delay`

For the `igetdevaddr`, `igetintftype`, and `igetsesstype` functions to be supported with the TCP/IP Instrument Protocol, the remote address strings *must* follow the TCP/IP Instrument Protocol naming conventions — `gpib0`, `gpib1`, and so forth. For example:

```
gpib0,7
gpib1,7,2
gpib2
```

However, since the interface names at the remote server may be configurable, this is not guaranteed. Also note that the correct behavior of `iremote` and `iclear` depend on the correct address strings being used.

Finally, note that when **iremote** is executed over the TCP/IP Instrument Protocol, **iremote** will also send the LLO (local lockout) message in addition to placing the device in the remote state.

Any of the following functions may timeout over LAN, even those functions which cannot timeout over local interfaces. (See the "Using Timeouts with LAN" section later in this chapter for more details.) These functions all cause a request to be sent to the server for execution.

- All GPIB/HP-IB 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**

The following SICL functions perform as follows with LAN-gatewayed sessions:

`idrvrversion`	Returns the version numbers from the server.
`iwrite`, `iread`	`actualcnt` may be reported as 0 when some bytes were transferred to or from the device by the server. This can happen if the client times out while the server is in the middle of an I/O operation.

LAN-gatewayed Session Example

The following example programs open an HP-IB device session via a LAN-to-HPIB gateway. Note that these examples are the same as the first example in the "Using HP SICL with HP-IB" chapter, only the addresses passed to the `iopen` calls are modified. The addresses used in these examples assume the machine with hostname `instserv` is acting as a LAN-to-HPIB gateway.

C Example:

```
/* landev.c
   This example program sends a scan list to a switch and while
   looping closes channels and takes measurements. */
#include <sicl.h>
#include <stdio.h>

main()
{
  INST dvm;
  INST sw;

  double res;
  int i;

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

  /* Open the multimeter and switch sessions */
  dvm = iopen ("lan[instserv]:hpib,9,3");
  sw = iopen ("lan[instserv]:hpib,9,14");
  itimeout (dvm, 10000);
  itimeout (sw, 10000);

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

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

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

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

    /* Print the results */
    printf ("Result is %f\n",res);
    /*Trigger to close channel*/
    iprintf (sw, "TRIG\n");
  }
  /* Close the multimeter and switch sessions */
  iclose (dvm);
  iclose (sw);
}
```

Visual BASIC Example:

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

Attribute VB_Name = "Module1"

Public Sub lanmain()
    Dim dvm As Integer, sw As Integer
    Dim nargs As Integer, I As Integer
    Dim res As Double
    Dim actual As Long
    Dim res1 As String

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

    'Open the multimeter and switch sessions
    dvm = iopen("lan[intserv]:hpib,9,3")
    sw = iopen("lan[intserv]:hpib,9,14")

    Call itimeout(dvm, 10000)
    Call itimeout(sw, 10000)

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

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

    For I = 1 To 4 Step 1
        nargs = iwrite(id, "MEAS:VOLT:DC?" + Chr$(10) + Chr$(0), 14, 1, actual)
        nargs = iread(id, res1, 1, &H0&, actual)

        MsgBox "Result is"
        MsgBox res1

        nargs = iwrite(id, "TRIG" + Chr$(10) + Chr$(0), 5, 1, actual)
    Next I

    Dim x As Integer
    x = iclose(dvm)
    x = iclose(sw)

    Exit Sub
```

```
ErrorHandler:

'  Display the error message in the txtResponse TextBox.
   txtResponse.Text = "*** Error : " + Error$
   MsgBox txtResponse.Text
'  Close the device session if iopen was successful.
   If id <> 0 Then
      iclose (id)
   End If

   Exit Sub
End Sub
```

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 the "Using Timeouts with LAN " section later 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 `I/O Config` utility from the `HP I/O Libraries` program group. See Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows* for information on running `I/O Config`.

The following are examples of LAN interface addresses:

`lan`	A LAN interface address using the interface name `lan`.
`30`	A LAN interface address using the logical unit `30`. (Note that `30` is the default logical unit for LAN.)

HP SICL Function Support with LAN Interface Sessions

The following SICL functions are *not* supported over LAN interface sessions and will return `I_ERR_NOTSUPP`:

- All HP-IB specific functions
- All serial specific functions
- All formatted I/O routines
- `iwrite`
- `iread`
- `ilock`
- `iunlock`
- `isetintr`
- `itrigger`
- `ixtrig`
- `ireadstb`
- `isetstb`
- `imapinfo`
- `ilocal`
- `iremote`

The following SICL functions perform as follows with LAN interface sessions:

`iclear`	Performs no operation, returns `I_ERR_NOERROR`.
`ionsrq`	Performs no operation against LAN gateways for SICL, returns `I_ERR_NOERROR`.
`ionintr`	Performs no operation, returns `I_ERR_NOERROR`.
`iabort`	Performs no operation, returns `I_ERR_NOERROR`.
`igetluinfo`	This function returns information about local interfaces only. It does not return information about remote interfaces that are being accessed via a LAN-to-instrument_interface gateway.

Using Locks and Multiple Threads over LAN

If two or more threads are accessing the same device or interface using two or more different sessions over LAN, and they are using SICL locks to synchronize access, the following situations may cause timeouts to occur or may hang an application which does not use timeouts. The common idea in all of these scenarios is that all threads which are using 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**. Hence, the following scenarios should be avoided:

- Using a hostname to identify the remote host in one call to **iopen** while using an alias or IP address to identify the same host in another call to **iopen**.
- Using a device symbolic name in one call to **iopen** (such as **"dmm"**, where **"dmm"** equals **"hpib,1"**) while using the fully specified device name (such as **"hpib,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 **"hpib"**) in another.
- Using **igetintfsess** to open an interface session (which internally uses the logical unit to identify the remote interface) while opening the interface with its symbolic name for another session.

You can avoid each of the previous situations 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.

Note that if any thread is using **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**, and so forth) to a device already locked via a different session from a different thread within the same process.

Note that 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).

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

When the client sends an I/O request to the server, the timeout value specified with `itimeout`, or the SICL default, is passed with the request. The server will use 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, then the server will send a reply to the client which indicates that a timeout occurred, and the SICL call made by the application will return `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, then 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 **I/O Config** utility (see the next subsection). Once **ilantimeout** is called by the application, the automatic LAN timeout adjustment described in the next subsection is turned off. See the *HP SICL Reference Manual* for details of the **ilantimeout** and **ilangettimeout** functions.

Note that 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 in special circumstances only and should be used with extreme caution. For more information about this timeout value, see the section, "Using the No-Wait Value," under the **ilantimeout** function in the *HP SICL Reference Manual.*

Default LAN Timeout Values

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

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

NOTE

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 note that `ilantimeout` is *per process*. That is, all sessions which are going out over the network are affected when `ilantimeout` is called.

If the application has *not* called the **ilantimeout** function, then the 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 **I/O 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, do the following:

1. Exit any LAN applications for SICL which you want to reconfigure.
2. Run the **I/O Config** utility. (Double-click the **I/O Config** icon in the **HP I/O Libraries** program group.) Change the Server Timeout and/or Client Timeout Delta value(s).
3. Restart the LAN applications for SICL.

Timeouts in Multi-threaded Applications

If you need to manually set the client side timeout in an application using multiple threads, note 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 will be 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`).

Also, the use of the `ilantimeout` No-Wait Value for certain special cases is described under the `ilantimeout` function in the *HP SICL Reference Manual*. If the no-wait value is used and multiple threads are attempting I/O over the LAN, the I/O operations using the no-wait option will wait for access to the LAN for 2 minutes. If another thread is using the LAN interface for greater than 2 minutes, the no-wait operation will timeout.

Timeout Configurations to Be Avoided

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

The SICL timeout used by the server should generally be less than infinity. Having the LAN server wait less than forever allows the LAN server to detect clients that have died abruptly or network problems and subsequently release resources associated with those clients, such as locks. Using the smallest possible value for your application will maximize the server's responsiveness to dropped connections, including the client application being terminated abnormally. Using a value less than infinity is made easy for application developers due to the Server Timeout configuration value set via the `I/O Config` utility. Even if your application uses the SICL default of infinity, or if `itimeout` is used to set the timeout to infinity, by setting the Server Timeout value to some reasonable number of seconds, the server will be allowed to timeout and detect network trouble if it has occurred and release resources.

Application Terminations and Timeouts

If an application is killed while in the middle of a SICL operation which is performed at the LAN server, the server will continue to try the operation until the server's timeout is reached. By default, the LAN server associated with an application using a timeout of infinity which is killed may not discover that the client is no longer running for 2 minutes. (If you are using a server other than the LAN server on HP-UX, Windows 95, Windows NT, or the HP E2050, check that server's documentation for its default behavior.)

If `itimeout` is used by the application to set a long timeout value, or if both the LAN client and LAN server are configured to use infinity or a long timeout value, then the server may appear "hung." If this situation is encountered, the LAN client (via the Client Timeout Delta value set via the `I/O 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 killing the LAN server that is running. Note that the latter procedure will affect all clients connected to the server. See the LAN section in Chapter 9, "Troubleshooting Your HP SICL Program," for more details. Also see the documentation of the server you are using for the method to be used to reset the server.

Function Name	Action
ilantimeout	Sets LAN timeout value
ilangettimeout	Returns LAN timeout value
igetgatewaytype	Indicates whether the session is via a LAN gateway

9

Troubleshooting Your HP SICL Program

Troubleshooting Your HP SICL Program

This chapter contains a description of SICL error codes. It also provides help for troubleshooting common problems with SICL and:

- Windows 95
- WIN16 Programs on Windows 95 and Windows 3.1
- Windows 3.1
- Windows NT
- RS-232
- GPIO
- LAN Client and Server

HP SICL Error Codes

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

- On Windows 95 or Windows 3.1, start the **Message Viewer** utility by double-clicking on the icon either in the **HP I/O Libraries** program group for Windows 95, or in the **HP SICL** program group for Windows 3.1. You may want to iconify the utility during execution of your program. However, you must always start it before you execute a program in order for messages to be logged.
- On Windows NT, SICL logs internal messages as Windows NT events that you can view by starting the **Event Viewer** utility in the **Administrative Tools** group. 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 HP-IB driver).

If you are programming in C, you may also use `ionerror` to install your own custom error handler. Your error handler can call `igeterrstr` with the given error code, and the corresponding error message string will be returned.

See either the "Error Handlers in C" or "Error Handlers in Visual BASIC" section in Chapter 4, "Programming with HP SICL," for more information on installing error handlers.

The following table contains an alphabetical summary of SICL error codes and messages.

SICL Error Codes and Messages

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

SICL Error Codes and Messages (continued)

Error Code	Error String	Description
I_ERR_NESTEDIO	Nested I/O	Attempt to call another SICL function when current SICL function has not completed [WIN16]. More than one I/O operation is prohibited.
I_ERR_NOCMDR	Commander session is not active or available	Tried to specify a commander session when it is not active, available, or does not exist.
I_ERR_NOCONN	No connection	Communication session has never been established, or connection to remote has been dropped.
I_ERR_NODEV	Device is not active or available	Tried to specify a device session when it is not active, available, or does not exist.
I_ERR_NOERROR	No Error	No SICL error returned; function return value is zero [0].
I_ERR_NOINTF	Interface is not active	Tried to specify an interface session when it is not active, available, or does not exist.
I_ERR_NOLOCK	Interface not locked	An **iunlock** was specified when device wasn't locked.
I_ERR_NOPERM	Permission denied	Access rights violated.
I_ERR_NORSRC	Out of resources	No more system resources available.
I_ERR_NOTIMPL	Operation not implemented	Call not supported on this implementation. The request is valid, but not supported on this implementation.
I_ERR_NOTSUPP	Operation not supported	Operation not supported on this implementation.
I_ERR_OS	Generic O.S. error	SICL encountered an operating system error.
I_ERR_OVERFLOW	Arithmetic overflow	Arithmetic overflow. The space allocated for data may be smaller than the data read.
I_ERR_PARAM	Invalid parameter	The constant or parameter passed is not valid for this call.
I_ERR_SYMNAME	Invalid symbolic name	Symbolic name passed to **iopen** not recognized.
I_ERR_SYNTAX	Syntax error	Syntax error occurred parsing address passed to **iopen**. Make sure that you have formatted the string properly. White space is not allowed.
I_ERR_TIMEOUT	Timeout occurred	A timeout occurred on the read/write operation. The device may be busy, in a bad state, or you may need a longer timeout value for that device. Check also that you passed the correct address to **iopen**.
I_ERR_VERSION	Version incompatibility	The **iopen** call has encountered a SICL library thet is newer than the drivers. Need to update drivers.

Subsequent Execution of SICL Application Fails

If you terminate a program using the Task Manager, or if a program has an abnormal termination, then some drivers may not unload from memory. This could cause subsequent attempts to execute your I/O program to fail. To recover from this situation, you must restart (reboot) Windows 95.

Common Problems with WIN16 Programs on Windows 95 and Windows 3.1

Subsequent Execution of SICL Application Gives Strange Behavior

Check that `_siclcleanup` for C, or `siclcleanup` for Visual BASIC, is called at the end of the program and at any other possible program exit-point (at error handlers, and so forth).

General Protection Fault Occurs When Interrupt, SRQ, or Error Handler Called

Check that the interrupt or error handler routine was declared with the `SICLCALLBACK` modifer. Also, make sure that compiler options to generate prolog code for exported functions were selected. If you are using the QuickWin feature of Microsoft compilers, you must also use the `_loadds` modifier in the handler declaration.

General Protection Fault When Calling SICL Formatted I/O Routine

Verify that all pointer parameters passed to SICL formatted I/O routines are declared as `_far`, or that the compiler large memory model option is selected.

Reference to Undefined Function or Array

Visual BASIC gives this message when a call is made to a function that is not defined in a program. If you get this message when you try to call a SICL routine, then you need to add the `SICL4.BAS` file to your Project for Visual BASIC 4.0, or the `SICL.BAS` file to your Project for Visual BASIC 3.0 or earlier.

Do this by selecting `File | Add` from the Visual BASIC menu. The `SICL4.BAS` and `SICL.BAS` files are located in the `VB` subdirectory under the SICL base directory (for example, `C:\SICL\VB` if you installed SICL in the default directory).

General Protection Fault Occurs When `iwrite`, `iread`, or `ivscanf` is Called from Visual BASIC

Make sure that all strings used as buffers for `iread` and `ivscanf` are fixed length strings that are large enough to accommodate the data to be read in.

I_ERR_NESTED_IO Occurs

A subsequent SICL function call has been made before a previous call completed. In order to allow other WIN16 applications to execute while a SICL application is running, SICL may temporarily suspend execution in the middle of a SICL call while waiting for a slow HP-IB transaction to complete. Without this feature, your Windows system would be locked up until the transaction completes.

However, because Windows is an event-/message-driven operating system, it is possible that the SICL application would receive a message instructing it to initiate another SICL call before the first one completes. This will result in a SICL error. Your program must be designed so that this situation does not occur.

Unresolved SICL Externals When Building a SICL Application

Check that you are linking to the correct import libraries. Refer to Chapter 3, "Building an HP SICL Application."

Can't Find "llibxxxx" When Building a SICL Application

You probably did not load the large memory model libraries when you installed your compiler software. Re-run the setup program for your compiler and specifically request that large memory model libraries be installed.

Common Problems with Windows NT

Program Appears to Hang and Cannot Be Killed

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 killed.

To kill the application, click on 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 will appear. Press the **End Task** button. The application is now killed.

Formatted I/O Using %F Causes Application Error

Verify that you are using **$(cvarsdll)** when compiling your application, and either **$(guilibsdll)** for Windows applications or **$(conlibsdll)** for console applications when linking your application.

Also note that the **%F** format character for **iprintf** and **iscanf** only works with languages that use **MSVCRT20.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. Therefore, they cannot use **%F** at all.

Common Problems with RS-232

Unlike HP-IB, special care must be taken to ensure that RS-232 devices are correctly connected to your computer. Verifying your configuration first can save many wasted hours of debugging time.

No Response from Instrument

Check to make sure that the RS-232 interface is configured to match the instrument. Check the Baud Rate, Parity, Data Bits, and Stop Bits.

Also make sure that you are using the correct cabling. Refer to the "RS-232 Cables and HP Instruments" appendix in the *HP I/O Libraries Installation and Configuration Guide*, as well as the *RS-232 Cables Addendum* included in your HP I/O Libraries product package for more information on correct cabling.

If you are sending many commands at once, try sending them one at a time either by inserting delays, or by single-stepping your 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 the following:

- Flow control settings match
- Full/half duplex for 3-wire connections
- Cabling is correct for hardware handshaking

Common Problems with GPIO

Because the GPIO interface has such flexibility, most initial problems come from cabling and configuration. There are many configuration fields in the **I/O 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 in it, and you often must solder your own plug to at least one end. It is important to have the hardware configuration under control before you begin troubleshooting your software.

If you are porting an existing HP 98622 application, the hardware task is simplified. The cable connections are the same, and many **I/O Config** fields closely approximate HP 98622 DIP switches. If yours is a new application, someone on the project with good hardware skills should become familiar with the HP E2075 cabling and handshake behavior. In either case, it is important to read the *HP E2074/5 GPIO Interface Installation Guide.*

The following are some GPIO-specific reasons for certain SICL errors. Keep in mind that many of these 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.) Such general causes are discussed earlier in this book. The following discussion highlights the causes of errors that relate directly to the HP E2075 GPIO interface.

Bad Address (for iopen)

This means the same thing for GPIO as for any interface. It indicates that the **iopen** did not succeed because the specified address (symbolic name) does not correspond to a correctly configured entry in **I/O Config**. This is mentioned here because the GPIO has more configuration fields (and thus more chances for mistakes) than any other interface.

If your **iopen** fails, make sure that the interface was properly configured. The **I/O Config** utility establishes an entry for the GPIO card in your Windows 95 or Windows NT registry. You are strongly encouraged to let **I/O Config** handle all registry maintenance for SICL. However, there is nothing which prohibits you from editing registry entries manually. If you manually change the registry and enter an improper configuration value, then the failed **iopen** may send a diagnostic message to the **Message Viewer** (on Windows 95) or **Event Viewer** (on Windows NT) utility. For example:

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

In such circumstances, you need to correct the configuration data in the registry. The recommended procedure is to use **I/O Config**, remove the problematic interface name, and create a **Configured Interface** with legal values selected from the **I/O Config** utility's dialog boxes.

Operation Not Supported

The HP 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 HP 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 that you wanted PSTS checks for read/write operations, and a false state of the PSTS line was detected. Enabling and disabling PSTS checks is done with the command:

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

If the check seems to be reporting the wrong state of the PSTS line, then correct the PSTS polarity bit via the `I/O Config` utility. If the PSTS check is functioning properly and you get this error, then some problem with the cable or the peripheral device is indicated.

Bad Parameter

This error has the same meaning for GPIO as for any interface. However, one case may be less obvious than typical parameter passing errors. 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 Problems with HP SICL over LAN (Client and Server)

NOTE

Both the LAN client and server may log messages to the **Message Viewer** (on Windows 95) or **Event Viewer** (on Windows NT) utility under certain conditions, whether an error handler has been registered or not.

Before SICL over LAN can be expected to function, the client must be able to talk to the server over the LAN. Use the following techniques to determine whether the problem you are experiencing is a general network problem, or is specific to the LAN software provided with SICL:

- If your application is unable to open a session to the LAN server for SICL, the first diagnostic to try is the **ping** utility. This command allows you to test general network connectivity between your client and server machines. Using **ping** looks something like the following:

```
>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
```

  Where each line after the **Pinging** line is an example of a packet successfully reaching the server.

However, if `ping` is unable to reach the host, you will see a message similar to the following:

```
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.
```

This indicates that the client was unable to contact the server. In this situation you should contact your network administrator to determine what is wrong with the LAN. Once the LAN problem has been corrected, you can then retry your SICL application over LAN.

- Another tool which can be used to determine where a problem might reside is the `rpcinfo` utility on an HP-UX workstation or other UNIX workstation. This tool tests whether a client can make an RPC call to a server. The first `rpcinfo` option to try is `-p`, which will print a list of registered programs on the server:

```
> rpcinfo -p instserv
program verses proto   port
100001    1   udp   1788  rstatd
100001    2   udp   1788  rstatd
100001    3   udp   1788  rstatd
100002    1   udp   1789  rusersd
100002    2   udp   1789  rusersd
395180    1   tcp   1138
395183    1   tcp   1038
```

Several lines of text will likely be returned, but the ones of interest are the lines for programs `395180`, which is for the SICL LAN Protocol, and `395183`, which is for the TCP/IP Instrument Protocol (the port numbers will vary). If the line for program `395180` or `395183` is not present, your LAN server is likely misconfigured. Consult your server's documentation, correct the configuration problem, and then retry your application.

- The second `rpcinfo` option which can be tried is `-t`, which will attempt to execute procedure 0 of the specified program. You should see a line similar to the following.

  For the SICL LAN Protocol:

  ```
  > rpcinfo -t instserv 395180
  program 395180 version 1 ready and waiting
  ```

  For the TCP/IP Instrument Protocol:

  ```
  > rpcinfo -t instserv 395183
  program 395183 version 1 ready and waiting
  ```

  If you do not see one of the above, your server is likely misconfigured or not running. Consult your server's documentation, correct the problem, and then retry your application. See the `rpcinfo(1M)` man page for more information.

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 that 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 that 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 **I/O Config** utility. See the "Using Timeouts with LAN" section in Chapter 8, "Using HP SICL with LAN," for more information.

`iopen` Fails - Other Failures

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

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 `I/O Config` utility. Also ensure that the LAN client timeout is large enough if you used `ilantimeout`. See the "Using Timeouts with LAN" section in Chapter 8, "Using HP 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 that 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 you are using `ilantimeout`, you must determine and set the LAN timeout manually. Otherwise, ensure that the Client Timeout Delta configuration value is large enough (via the `I/O Config` utility). See the "Using Timeouts with LAN" section in Chapter 8, "Using HP 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. Old LAN server connections for SICL from previous clients may not have terminated properly. Consult your server's troubleshooting documentation and follow its instructions for cleaning-up any old 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. Simply try running the SICL LAN application again.

`rpcinfo` Does Not List `395180` or `395183`

An `rpcinfo` fails to indicate that program `395180` (SICL LAN Protocol) or `395183` (TCP/IP Instrument Protocol) is available on the server. Did you start the LAN server? If not, do so. (See the "Starting the LAN Server" section of Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows*.) If so, try the `rpcinfo` query again after a few seconds to ensure that the LAN server had time to register itself.

`iopen` Fails

An `iopen` fails when you run your application, but `rpcinfo` indicates that the LAN server is ready and waiting. Ensure that the requested interface has been configured on the server. See Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows* for information on using the `I/O Config` utility to configure interfaces for SICL.

LAN Server Appears "Hung"

The LAN server appears hung (possibly due to a long timeout being set by a client on an operation which will never succeed). Login to the LAN server and stop the hung LAN server process. (To stop the LAN server, see the "Stopping the LAN Server" section of Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows*.) Note that this will affect all connected clients, even those which may still be operational. If informational logging has been enabled using the `I/O Config` utility, then connected clients can be determined by log entries in either the `Message Viewer` (on Windows 95) or `Event Viewer` (on Windows NT) utility.

`rpcinfo` Fails - `can't contact portmapper`

An `rpcinfo` returns the message `rpcinfo: can't contact portmapper: RPC_SYSTEM_ERROR - Connection refused`. Ensure that the LAN server is running. If not, start it. If so, stop the currently running LAN server and restart it. Use Ctrl-Alt-Del to get a task list. Ensure that both **`LAN Server`** and **`Portmap`** are not running before restarting the LAN server. (See the "Starting the LAN Server" and the "Stopping the LAN Server" sections of Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows*.)

`rpcinfo` Fails - `program 395180 is not available`

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

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

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

Mouse Hung When Stopping LAN Server

If after attempting to stop a LAN server via either Ctrl-C or the Windows 95 X-button (the "kill" button in the upper-right hand corner of a Windows 95 window), the mouse may appear to be hung. Hit any keyboard key and the LAN server will stop and the mouse will again be operational.

10

More HP SICL Example Programs

More HP SICL Example Programs

This chapter contains additional example programs that help show you how to develop your SICL applications. The example programs are:

- An example C program for oscilloscopes
- An example Visual BASIC program for oscilloscopes

Example C Program for Oscillosopes

This C example programs an oscilloscope (such as an HP 54601), uploads the measurement data, and instructs the scope to print its display to a ThinkJet printer. This example program uses many SICL features and illustrates some important C and Windows programming techniques for SICL. An overview of the program follows the sections on building the program for Windows 3.1 and Windows NT.

The oscilloscope example files are located in the `C\SAMPLES\SCOPE` subdirectory under the SICL base directory (for example, `C:\SICL\C\SAMPLES\SCOPE` if you installed SICL in the default location). The subdirectory contains the source program and a number of files to help you build the example with specific compilers, depending on which Windows environment you are using.

`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.
`MSCSCOPE.MAK`	Windows 3.1 makefile for Microsoft C and Microsoft SDK compilers.
`VCSCOPE.MAK`	Windows 3.1 project file for Microsoft Visual C++.
`VCSCP32.MAK`	Windows 95 or Windows NT (32-bit) project file for Microsoft Visual C++.
`VCSCP16.MAK`	Windows 95 or Windows 3.1 (16-bit) project file for Microsoft Visual C++.
`QCSCOPE.MAK`	Windows 3.1 project file for Microsoft QuickC for Windows.

`BCSCOPE.IDE`	Windows 3.1 project file for Borland C Integrated Development Environment.
`BCSCP32.IDE`	Windows 95 or Windows NT (32-bit) project file for Borland C Integrated Development Environment.
`BCSCP16.IDE`	Windows 95 (16-bit) project file for Borland C Integrated Development Environment.

Building a 16-bit C Program for Windows 95 or Windows 3.1

This section explains how to create the project file for this example using Microsoft Visual C. The following procedure summarizes many of the considerations discussed earlier in this guide. An overview of this example program is provided later in this chapter.

You may also load the makefile directly from the `C\SAMPLES\SCOPE` subdirectory, if you desire. If you are using another language tool, choose the corresponding project file or makefile from the `C\SAMPLES\SCOPE` subdirectory.

To compile and link the example program with Microsoft Visual C, follow these steps:

1. Select **`Project | Open`** from the menu, and enter a name (with the path of your working directory) for the project in the dialog box. Also select **`Windows Application`** as the project type. Click on the **`OK`** button.
2. The **`Edit`** dialog box will now be displayed. Double-click on the source file `SCOPE.C` to add it to the project. Also add `SICL16.LIB` and `MSAPP16.LIB` from the SICL `C` directory (for example, `C:\SICL\C` if you installed SICL in the default location). Click on the **`Close`** button.
3. Select **`Options | Directories`** from the menu and add the SICL `C` directory (for example, `C:\SICL\C` if you installed SICL in the default location) to the end of the paths listed in the **`Include`** edit box. Be sure to add a semicolon before the SICL path. Click the **`OK`** button.
4. Select **`Options | Project`**. Click on the **`Compiler`** button, then select **`Memory Model`** from the **`Category`** list. Click on the **`Model`** list arrow to display the model options, and select **`Large`**. Click on **`OK`** to close the Compiler dialog box.
5. 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 be opened. 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.

Building a 32-bit C Program for Windows 95 or Windows NT

This section explains how to create the project file for this example using Microsoft Visual C. The following procedure summarizes many of the considerations discussed earlier in this guide. An overview of this example program is provided in the next section.

You may also load the makefile directly from the **C\SAMPLES\SCOPE** subdirectory, if you desire. If you are using another language tool, choose the appropriate project file or makefile from the **C\SAMPLES\SCOPE** subdirectory.

To compile and link the example program with Microsoft Visual C, follow these steps:

1. Select **File | New** from the menu and select **Project** from the list box that appears. Then click on **OK**.
2. The **New Project** dialog box is now displayed. Type the name you want for the project in the edit box labeled **Project Name**. Then select **Application** from the **Project Type** list box. Select the directory location for the project in the **Directory** list box. Then click on the **Create** button.
3. The **Project Files** dialog box is now displayed. Double-click on 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 (for example, **C:\SICL\C** if you installed SICL in the default location). Then press the **Close** button.
4. Select **Project | Settings** from the menu and click on 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. Click on **OK**.
5. Select **Tools | Options** from the menu and click on the **Directories** button in the **Options** dialog box. Select **Include Files** from the **Show Directories for:** list box and click on the **Add** button. Then type in **\SICL\C** and click on **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.

C Program Overview

You may want to view the program with an editor as you read through this section. (The entire oscilloscope example program is not listed here because of its length.) This example program is designed to illustrate particular SICL features and programming techniques — it is not meant to be a robust Windows application.

Refer to the *HP SICL Reference Manual* or the SICL online **Help** for more detailed information on the SICL features used in this example program.

Custom Error Handler

The oscilloscope program defines a custom error handler that will be called whenever an error occurs during a SICL call. The handler is installed using **ionerror** before any other SICL function call is made, and will be used for all SICL sessions created in the program.

```
void SICLCALLBACK my_err_handler(INST id, int error)
{
    ...
    sprintf(text_buf[num_lines++],
            "session id=%d, error = %d:%s", id, error, igeterrstr(error));
    sprintf(text_buf[num_lines++], "Select 'File | Exit' to exit program!");
    ...

    // If error is from scope, disable I/O actions by graying out menu picks.
    if (id == scope) {
        ... code to disallow further I/O requests from user

    }
}
```

The error number is passed to the handler, and **igeterrstr** is used to translate the error number into a more useful description string. If desired, different actions can be taken depending on the particular error or **id** that caused the error.

Locks

SICL allows multiple applications to share the same interfaces and devices. Different applications may access different devices on the same interface, or may alternately access the same device (a shared resource). If your program will be executing along with other SICL applications, you may wish 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 the 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 others
    iunlock(scope);

    ... non-SICL code
}
```

Lock the interface or device with **ilock** before critical sections of code, and release the resource with **iunlock** at the end of the critical section. Using **ilock** on a device session prevents any other device session from accessing the particular device. Using **ilock** on an interface session prevents any other session from accessing the interface and any device connected to the interface. See the description of **isetlockwait** in the *HP SICL Reference Manual* to determine what actions can be taken when a SICL call in your code attempts to access a resource that is locked by another session.

Formatted I/O

SICL provides extensive formatted I/O functionality to help facilitate communication of I/O commands and data. The example program uses just 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 output 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 scope, we 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 scope waveform data, the string **"%#wb\n"** is used. The **wb** indicates that it 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 has been read.

Interface Sessions

Sometimes it may be necessary to control the HP-IB bus directly instead of using SICL commands that do it for you. 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 scope 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 out on the bus to tell the printer to listen and the scope to send its data. The **igpibatnctl** function directly controls the state of the ATN signal on the bus.

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

    hpibintf = igetintfsess(id);
    ...

    // tell scope 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(hpibintf,cmd,length);
    igpibatnctl(hpibintf,0);

    ...
}
```

SRQs and `iwaithdlr` Many instruments are capable of using the service request, or SRQ, signal on the HP-IB bus to signal the controller that an event has occurred. If an application wishes to respond to SRQs, an SRQ handler must be installed with the `ionsrq` call. All SRQ handlers will be called whenever an SRQ occurs.

In the example handler, the scope status is read to verify that the scope asserted SRQ, and then the SRQ is cleared and a status message is displayed. If the scope 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 scope to print its display, the scope is set to assert SRQ when printing is finished. While the scope 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 scope to print and send an SRQ is next, then the call to **iwaithdlr**, with a timeout value of 30 seconds. When the scope 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 HP 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!") ;
    ...
}
```

Nested I/O and 16-bit Windows

WIN16 programs must be designed so that a new SICL call cannot be initiated until the previous call completes. In this program, it would be possible for the user to select another action from the program menu that required a SICL call before the previous action completed. To prevent this, any action that uses SICL functions first disables all other actions by using Windows commands to gray out and disable other I/O menu items (see the **enable_io_menu_items** function). These menu items are then re-enabled after the desired SICL calls have completed.

```
void print_disp (INST id)
{
    ... non-SICL code

    // do this before making all SICL calls
    enable_io_menu_items(FALSE);

    ... SICL I/O code

    // do this after making all SICL calls
    enable_io_menu_items(TRUE);
}
```

Example Visual BASIC Program for Oscillosopes

This Visual BASIC example program uses 16-bit SICL to get and plot waveform data from an HP 54601A (or compatible) oscilloscope. This routine is called each time the **cmdGetWaveform** command button is clicked. This example program uses many SICL features and illustrates some important Visual BASIC and Windows programming techniques for SICL. An overview of the program follows the section on loading and running the program for Windows 3.1 and Windows 95.

The oscilloscope example files are located in the **VB\SAMPLES\SCOPE** subdirectory under the SICL base directory (for example, **C:\SICL\VB\SAMPLES\SCOPE** if you installed SICL in the default location). The files provided are:

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

Loading and Running the Visual BASIC Program for Windows 95, Windows NT, and Windows 3.1

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

1. Connect an HP 54601A scope to your interface.
2. Run Visual BASIC.
3. Open the project file **SCOPE.MAK** by selecting **File | Open Project** from the Visual BASIC menu.
4. Edit the **SCOPE.FRM** file to set the **scope_address** constant to the address of your scope. To do this:
   a. Select **Window | Procedures** from the Visual BASIC menu. A View Procedure dialog box will appear.
   b. Select **SCOPE.FRM** from the Modules list box and **(declarations)** from the Procedures list box. Then click **OK**.
   c. Edit the following line so that the address is set to the address of your scope:

```
>>  Const scope_address = "hpib7,7"
```

5. Run the program by pressing either the (F5) key or the **RUN** button on the Visual BASIC Toolbar.
6. Press the **Waveform** button to get and display the waveform.
7. Press the **Integral** button to calculate and display the integral.

Note that after performing the previous steps, you can create a standalone executable (**.EXE**) version of this program by selecting **File | Make EXE File** from the Visual BASIC menu.

Visual BASIC Program Overview

You will want to view the program with an editor as you read through this section. (The oscilloscope example program is not listed here because of its length.) Refer to the *HP SICL Reference Manual* or the SICL online `Help` for more detailed information on the SICL features used in this example program.

`cmdGetWave-form_Click` — Subroutine that gets called when the `cmdGetWaveform` command button is pressed. The command button is labeled `Waveform`.

`On Error` — This Visual BASIC statement enables an error handling routine within a procedure. In this example, an error handler is installed starting at label `ErrorHandler` within the `cmdOutputCmd_Click` subroutine. The error handling routine will be called any time an error occurs during the processing of the `cmdGetWaveform_Click` procedure. Note that SICL errors are handled in the same way that Visual BASIC errors are handled with the `On Error` statement.

`cmdGetWave-form.Enabled` — Notice how 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` — Next, an `iopen` call is made to open a device session for the scope. The device address for the scope is contained in the `scope_address` string. Note that, in this example, the default address is `"hpib7,7"`. The interface name `"hpib7"` is the name given to the interface with the `I/O Config` utility. The bus (primary) address of the scope follows, in this case 7. You will probably want to change the `scope_address` string to specify the correct address for your configuration.

`igetintfsess` — Next, `igetintfsess` is called to return an interface session *id* for the interface to which the scope instrument is connected. This interface session will be used by the following `iclear` call to send an interface clear to reset the interface.

`iclear` — The `iclear` function is called to reset the interface.

`itimeout` Next, `itimeout` is called to set the timeout value for the scope's device session to 3 seconds.

`ivprintf` The `ivprintf` function is called four times to set up the scope and then request the scope's preamble information. Notice in each case how `Chr$(10)` is appended to the format string passed as the second argument to `ivprintf`. This tells `ivprintf` to flush the formatted I/O write buffer after writing the string specified in the format string. Also, notice how `0&` is used to specify a NULL pointer for the third argument to `ivprintf`. A NULL pointer must be passed as the third argument since no argument conversion characters were specified in the format string for `ivprintf`.

`ivscanf` The `ivscanf` function is called to read the scope's preamble information into the preamble array. Note how the first element of the preamble array is passed as the third parameter to `ivscanf`. This passes the address of the first element of the preamble array to the `ivprintf` SICL function.

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

`ivprintf` Next `ivprintf` is called to prompt the scope for its waveform data. Again, notice how `Chr$(10)` is appended to the format string passed as the second argument to `ivprintf`. This tells `ivprintf` to flush the formatted I/O write buffer after writing the string specified in the format string. Also notice how `0&` is used to specify a NULL pointer for the third argument to `ivprintf`, since no additional arguments were specified in the format string.

`ivscanf` Next `ivscanf` is called to read in the scope's waveform. The waveform is read in as an arbitrary block of data. Note that the format string passed as the second parameter to `ivscanf` specifies a maximum of 4000 Integer values that can be read into the array. Also note how the first element of the waveform array is passed as the 3rd parameter to `ivscanf`. This passes the address of the first element of the waveform array to the SICL `ivscanf` function.

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

`iclose`	The **iclose** subroutine closes the **scope_id** device session for the scope as well as the **intf_id** interface session obtained with **igetintfsess**.
`cmdGetWave-form.Enabled`	Notice how the button that causes the **cmdGetWaveform_Click** routine to be called is re-enabled when execution inside **cmdGetWaveform_Click** is finished. This allows the program to get another waveform.
`Exit Sub`	This Visual BASIC statement causes the **cmdGetWaveform_Click** subroutine to be exited after normal processing has completed.
`errorhandler:`	This label specifies the beginning of the error handler that was installed for this subroutine. This handler gets called whenever a run-time error occurs.
`Error$`	This Visual BASIC function is called to get the error message for the error.
`iclose`	The **iclose** subroutine is called inside the error handler to close the **scope_id** device session for the scope as well as the **intf_id** interface session obtained with **igetintfsess**.
`cmdGetWave-form.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.

A

HP SICL System Information

HP SICL System Information

This appendix provides information on SICL software files and system interaction in Windows 95, Windows NT, and Windows 3.1. This information can be used as a reference for removing SICL from a system, if necessary.

Windows 95

File Location

All SICL files are installed in the base directory specified by the person who installs SICL, with the exception of several common files that Windows must be able to locate. On Windows 95, the following files are copied to the system subdirectory of the main Windows directory:

- SICL16.DLL
- SICLUT17.DLL
- SICLUT16.DLL
- SICL32.DLL
- SICLUT31.DLL
- SICLRPC.DLL

The Registry

SICL places the following key in the Windows 95 registry under HKEY_LOCAL_MACHINE:

```
Software\Hewlett-Packard\SICL\CurrentVersion
```

Also, if the LAN Server is configured, the following key will be created under HKEY_LOCAL_MACHINE if it didn't previously exist:

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

HP SICL Configuration Information

SICL configuration information is stored in the Windows 95 registry under the `Software\Hewlett-Packard\SICL\CurrentVersion` branch under `HKEY_LOCAL_MACHINE`.

Windows NT

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. The common files `SICL32.DLL` and `SICLRPC.DLL` are placed in the `SYSTEM32` directory, and the `HP341I32.SYS` and `HP074I32.SYS` are placed in the `SYSTEM32\DRIVERS` directory, under the main Windows directory.

The Registry

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

- `Software\Hewlett-Packard\SICL\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`

HP SICL Configuration Information

SICL configuration information is stored in the Windows NT registry under the `Software\Hewlett-Packard\SICL\CurrentVersion` branch under `HKEY_LOCAL_MACHINE`.

Windows 3.1

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. These common files, `SICL16.DLL`, `SICLUT16.DLL`, and `HPIB.DLL`, are placed in the Windows directory.

Use of `WIN.INI`

SICL modifies the `WIN.INI` file in the Windows directory during installation. A `[SICL]` section is added, and a `BASEDIR` declaration is made to record the location of the SICL directory on the system. This has no other impact on your Windows configuration.

Also note that SICL uses the `[ports]` section of the `WIN.INI` file to get configuration parameters for RS-232 interfaces. Both the `I/O Config` utility and the Windows Control Panel will modify this section.

HP SICL Configuration Database

The `I/O Config` utility creates and maintains a `SICL.INI` file containing all SICL configuration information. This file is located under the SICL base directory. Under normal circumstances, this file should only be modified using the `I/O Config` utility.

B

Porting from the HP 82335 Command Library

Porting from the HP 82335 Command Library

The following table provides a cross-reference between HP 82335 Command Library commands and SICL function calls. It can be used as an aid for porting programs written for the older-style calls to using SICL calls instead.

Remember to add **iopen**, **iclose**, and **_siclcleanup** calls to the program. Additionally, SICL provides other features (such as error handling) that you may wish to take advantage of, instead of doing a straight translation of your existing code.

Definitions:

—n/a—	Means "Not Available".
auto	Means "Happens Automatically" with no commands needed.

	82335 DOS Command Lib	82335 Windows DLL	SICL
SESSION CONTROL			
- Open/Close	auto	HpibOpen/HpibClose	iopen/iclose
- Lock/Unlock	—n/a—	—n/a—	ilock/iunlock
- Timeout	IOTIMEOUT	HpibTimeout	itimeout
- Error handler	—n/a—	—n/a—	ionerror
DATA OUTPUT			
- Unformatted	IOOUTPUTB	HpibOutputb	iwrite
- Formatted Strings	IOOUTPUTS	HpibOutputs	iprintf
- 488.2 Quoted Strings	—n/a—	—n/a—	iprintf
- ASCII Formatted Numbers			
- - Integer (NR1)	IOOUTPUT	HpibOutput	iprintf
- - Real number(NR2,NR3)	IOOUTPUT	HpibOutput	iprintf
- - Real array	IOOUTPUTA	HpibOutputa	iprintf
- Binary			
- - 16-bit Integers	IOOUTPUTB	HpibOutputb	iprintf
- - 32-bit Integers	IOOUTPUTB	HpibOutputb	iprintf
- - 32-bit Reals	IOOUTPUTB	HpibOutputb	iprintf
- - 64-bit Reals	IOOUTPUTB	HpibOutputb	iprintf
- 488.2 Binary			
- - #B, #Q, #H	—n/a—	—n/a—	iprintf
- - Arbitrary Block	IOOUTPUTAB	HpibOutputab	iprintf
- File Output	IOOUTPUTF	HpibOutputf	iprintf
DATA INPUT			
- Unformatted	IOENTERB	HpibEnterb	iread
- Formatted Strings	IOENTERS	HpibEnters	iscanf
- 488.2 Quoted Strings	—n/a—	—n/a—	iscanf
- ASCII Formatted Numbers			
- - Integer (NR1)	IOENTER	HpibEnter	iscanf
- - Real number(NR2,NR3)	IOENTER	HpibEnter	iscanf
- - Real array	IOENTERA	HpibEntera	iscanf

	82335 DOS Command Lib	82335 Windows DLL	SICL
DATA INPUT (cont'd)			
- Binary			
- -16-bit Integers	IDENTERB	HpibEnterb	iscanf
- -32-bit Integers	IOENTERB	HpibEnterb	iscanf
- -32-bit Reals	IOENTERB	HpibEnterb	iscanf
- -64-bit Reals	IOENTERB	HpibEnterb	iscanf
- 488.2 Binary			
- - #B, #Q, #H	—n/a—	—n/a—	iscanf
- - Arbitrary Block	IOENTERAB	HpibEnterab	iscanf
- File Input	IOENTERF	HpibEnterF	iscanf
DATA I/O CONTROL			
- Prompted Input	—n/a—	—n/a—	ipromptf
- Auto Byte Swap	YES	YES	YES
- DMA ON/OFF	IODMA	—n/a—	ihint
- EOI ON/OFF	IOEOI	HpibEoi	auto
- EOL string [length]	IOEOL	HpibEol	see note [2]
- Short T1 Capability	IOFASTOUT	HpibFastout	auto
- Set Match Char	IOMATCH	HpibMatch	itermchar
- Read termination reason	IOGETTERM	HpibGetterm	auto
INSTRUMENT CONTROL			
- CLEAR an instrument	IOCLEAR	HpibClear	iclear
- Put device in LOCAL	IOLOCAL	HpibLocal	ilocal or igpibrenctl
- Put device in REMOTE	IOREMOTE	HpibRemote	iremote or igpibrenctl
- TRIGGER a device	IOTRIGGER	HpibTrigger	itrigger
- Read stb (serial poll)	IOSPOLL	HpibSpoll	ireadstb

	82335 DOS Command Lib	82335 Windows DLL	SICL
BUS CONTROL			
- Send IFC	IOABORT	Hpibabort	iclear
- Make all dev's REMOTE	IOREMOTE	HpibRemote	igpibrenctl
- Make all dev's LOCAL	IOLOCAL	HpibLocal	igpibrenctl
- CLEAR all devices	IOCLEAR	HpibClear	iclear
- Lock device front panel	IOLLOCKOUT	HpibLlockout	igpibllo
- Send bus commands	IOSEND	HpibSend	igpibsendcmd
- TRIGGER bus	IOTRIGGER	HpibTrigger	itrigger
- Reset to Power up state	IORESET	HpibReset	auto
- Set interface bus addr.	IOCONTROL	HpibControl	igpibbusaddr
- Get interface status	IOSTATUS	HpibStatus	igpibbusstatus
- Set/Drop ATN	IOCONTROL	HpibControl	igpibatnctl
PARALLEL POLL			
- Configure Parallel Poll	IOPPOLLC	HpibPpollc	igpibppollconfig
- Unconfigure parallel poll	IOPPOLLU	HpibPpollu	igpibppollconfig
- Conduct a parallel poll	IOPPOLL	HpibPpoll	igpibppoll
MISC. CAPABILITIES			
- SRQ Support	IOPEN	—n/a—	ionsrq
- Interrupt support	—n/a—	—n/a—	ionintr/isetintr
- Wait for event	—n/a—(1)	—n/a—	iwaithdlr
NON-CONTROLLER			
- Respond to serial poll	IOREQUEST	HpibRequest	isetstb
- Respond to parallel poll	—n/a—	—n/a—	igpibppollresp
- Request Service	IOREQUEST	HpibRequest	isetstb
- Pass Control	IOPASSCTRL	HpibPassctl	igpibpassctl
- Take Control	IOTAKECTRL	HpibTakectl	auto
- Can be non-Sys Ctlr?	YES	YES	YES

Notes:

1. Can be done manually using IOSTATUS in a loop.
2. The LF^END EOL sequence (required by 488.2) can be added with `iprintf`.

C

Porting to Visual BASIC 4.0

Porting to Visual BASIC 4.0

This edition of this manual supports and shows how to program SICL applications in Visual BASIC version 4.0 or later. If you have SICL applications written in an earlier Visual BASIC version than version 4.0 (for example, version 3.0), you can easily port your SICL applications to Visual BASIC version 4.0.

Porting your SICL applications to Visual BASIC 4.0 is a simple matter of adding the **SICL4.BAS** declaration file (rather than the **SICL.BAS** file) to each project that calls SICL for WIN16 or WIN32 (16-bit or 32-bit) Visual BASIC 4.0 programs. There may also be changes in functions where you are 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, you only need to pass the **0&** null pointer because version 4.0 knows this is by reference:

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

Once you have added the **SICL4.BAS** declaration file to each project and removed **ByVal** keywords preceding null pointers for strings, your SICL applications will run correctly with Visual BASIC 4.0.

Glossary

Glossary

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

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

bus error handler
Programming code executed when a bus error occurs.

commander session
A session that communicates to the controller of this bus.

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

controller role
A computer acting as a controller communicating with a device.

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

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

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

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

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

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

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

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

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

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

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

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

non-controller role
A computer acting as a device communicating with a controller.

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

register
An address location that controls or monitors hardware.

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

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

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

symbolic name
A name corresponding to a single interface or device. This name uniquely identifies the interface or device on this controller. If there is more than one interface or device on the controller, each interface or 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). Note that multi-threaded applications are only supported with 32-bit SICL.

Index

D