Standard Instrument Control Library

User's Guide

Notice

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

Test & Measurement Systems Inc. (TAMS) shall not be liable for any errors contained in this document. *TAMS makes no warranties of any kind with regard to this document, whether express or implied. TAMS specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.* TAMS 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 Test & Measurement Systems Inc. product and replacement parts can be obtained from your local Sales and Service Office.

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 TAMS standard software agreement for the product involved.

Printing History

This is the seventh edition of the *Standard Insrtument Control Library User's Guide (for HP-UX)*. Note: on previous editions the Reference section was actually a separate manual. On previous versions, the manual was operating system specific.

Edition 1 - May 1994

Edition 2 - September 1994

Edition 3 - January 1995

Edition 4 - November 1995

Edition 5 - July 1998

Edition 6 - August 2001

Edition 7 - August 2003

Contents

SICL User's Guide
Edition 7

1. Introduction

SICL Overview 20
- SICL Features 20
- SICL User 21

Related Documents 22
- Other SICL Learning Products 22
- Other Documentation 23

2. Getting Started with SICL

Reviewing a SICL Program 27
Compiling and Linking a SICL Program 29
- Using Shared Libraries 29

Running an SICL Program 31
Getting Online Help 32
- Using Manual Pages 32

Where to Go Next 33

3. Using SICL

Including the sicl.h Header File 37
Opening a Communications Session 38
- Device Sessions 39
- Interface Sessions 40
- Commander Sessions 41

Sending I/O Commands 42
- Formatted I/O 42
- Non-Formatted I/O 52

Using Asynchronous Events 54
- SRQ Handlers 54
- Interrupt Handlers 55
- Temporarily Disabling/Enabling Asynchronous Events 55
- Asynchronous Events and Unix Signals 57
- Interrupt Handler Example 59

Using Error Handlers 61

Error Handler Example 62
Using Locking 64
Lock Actions 65
Locking in a Multi-user Environment 66
Locking Example 67

4. Using SICL with GPIB

Creating a Communications Session with GPIB 71
Communicating with GPIB Devices 72
Addressing GPIB Devices 72
SICL Function Support with GPIB Device Sessions 74
GPIB Device Session Example 75
Communicating with GPIB Interfaces 77
Addressing GPIB Interfaces 77
SICL Function Support with GPIB Interface Sessions 78
GPIB Interface Session Examples 79
Communicating with GPIB Commanders 85
Addressing GPIB Commanders 85
SICL Function Support with GPIB Commander Sessions 87
Summary of GPIB Specific Functions 88

5. Using SICL with GPIO

Creating a Communications Session with GPIO 91
Communicating with GPIO Interfaces 92
Addressing GPIO Interfaces 92
SICL Function Support with GPIO Interface Sessions 93
GPIO Interface Session Example 95
GPIO Interrupts Example 96
Summary of GPIO Specific Functions 98

6. Using SICL with VXI/MXI

Creating a Communications Session with VXI/MXI 103
Communicating with VXI/MXI Devices 104
Message-Based Devices 105
Register-Based Devices 109
Communicating with VXI/MXI Interfaces 118
Addressing VXI/MXI Interface Sessions 118
VXI/MXI Interface Session Example 120
Communicating with VME Devices 121
Declaring Resources 122
Mapping VME Memory 123
Reading and Writing to the Device Registers 125
Unmapping Memory Space 125
VME Interrupts 125
VME Example 126
Looking at SICL Function Support with VXI/MXI 130
Device Sessions 130
Interface Sessions 131
Using SICL Trigger Lines 132
Using i?blockcopy for DMA Transfers 135
Using VXI Specific Interrupts 138
Processing VME Interrupts Example 140
Summary of VXI/MXI Specific Functions 141

7. Using SICL with RS-232

Creating a Communications Session with RS-232 145
Communicating with RS-232 Devices 146
Addressing RS-232 Devices 146
SICL Function Support with RS-232 Device Sessions 148
RS-232 Device Session Example 149
Communicating with RS-232 Interfaces 150
Addressing RS-232 Interfaces 150
SICL Function Support with RS-232 Interface Sessions 151
RS-232 Interface Session Example 154
Summary of RS-232 Specific Functions 156

8. Using SICL with LAN

Overview of SICL LAN 163
LAN Software Architecture 165
SICL LAN Server 167
Considering LAN Configuration and Performance 168
Communicating with Devices over LAN 169
LAN-gatewayed Sessions 169
LAN Interface Sessions 176
Using Timeouts with LAN 178
LAN Timeout Functions 178
Default LAN Timeout Values 179
Timeout Configurations to Be Avoided 182
Application Terminations and Timeouts 183
Using Signal Handling with LAN 184
SIGIO Signals 184
SIGPIPE Signals 185
Summary of LAN Specific Functions 186

9. Troubleshooting Your SICL Program

Installing an Error Handler 189
Looking at Error Codes and Messages 190
Troubleshooting SICL 192
Compile and Link Errors 192
Run-time Errors 194
Troubleshooting SICL over LAN (Client and Server) 195
SICL LAN Client Problems and Possible Solutions 197
SICL LAN Server Problems and Possible Solutions 199
Troubleshooting SICL over RS-232 202
No Response from Instrument 202
RS-232 Port Allocation and HP-UX termio Functions 202
Data Received from Instrument is Garbled 203
Data Lost During Large Transfers 203
Troubleshooting SICL over GPIO 204
Bad Address (for iopen) 204
Operation Not Supported 205
No Device 206

Generic I/O Error 206
Bad Parameter 207
Where to Find Additional Information 208

10. SICL Language Reference

IABORT 212
IBLOCKCOPY 213
IBLOCKMOVEX 215
ICAUSEERR 217
ICLEAR 218
ICLOSE 219
IDEREFPTR 220
IFLUSH 221
IFREAD 223
IFWRITE 225
IGETADDR 226
IGETDATA 227
IGETDEVADDR 228
IGETERRNO 229
IGETERRSTR 231
IGETGATEWAYTYPE 232
IGETINTFSESS 233
IGETINTFTYPE 234
IGETLOCKWAIT 235
IGETLU 236
IGETLUINFO 237
IGETLULIST 238
IGETONERROR 239
IGETONINTR 240
IGETONSRQ 241
IGETSESSTYPE 242
IGETTERMCHR 243
IGETTIMEOUT 244
IGPIBATNCTL 245
IGPIBBUSADDR 246
IGPIBBUSSTATUS 247
IGPIBGETT1DELAY 249

IGPIBLLO 250
IGPIBPASSCTL 251
IGPIBPPOLL 252
IGPIBPPOLLCONFIG 253
IGPIBPPOLLRESP 254
IGPIBRENCTL 255
IGPIBSENDCMD 256
IGPIBSETT1DELAY 257
IGPIOCTRL 258
IGPIOGETWIDTH 263
IGPIOSETWIDTH 264
IGPIOSTAT 266
IHINT 269
IINTROFF 271
IINTRON 272
ILANGETTIMEOUT 273
ILANTIMEOUT 274
ILOCAL 277
ILOCK 278
IMAP 280
IMAPX 283
IMAPINFO 286
IONERROR 288
IONINTR 291
IONSRQ 293
IOPEN 294
IPEEK 296
IPEEKX8, IPEEKX16, IPEEKX32 297
IPOKE 298
IPOKEX8, IPOKEX16, IPOKEX32 299
IPOPFIFO 300
IPRINTF 302
IPROMPTF 312
IPUSHFIFO 313
IREAD 315
IREADSTB 317
IREMOTE 318
ISCANF 319

ISERIALBREAK 329
ISERIALCTRL 330
ISERIALMCLCTRL 334
ISERIALMCLSTAT 335
ISERIALSTAT 336
ISETBUF 340
ISETDATA 342
ISETINTR 343
ISETLOCKWAIT 351
ISETSTB 352
ISETUBUF 353
ISWAP 355
ITERMCHR 357
ITIMEOUT 358
ITRIGGER 359
IUNLOCK 361
IUNMAP 362
IUNMAPX 364
IVERSION 365
IVXIBUSSTATUS 366
IVXIGETTRIGROUTE 369
IVXIRMINFO 370
IVXISERVANTS 372
IVXITRIGOFF 373
IVXITRIGON 375
IVXITRIGROUTE 377
IVXIWAITNORMOP 379
IVXIWS 380
IWAITHDLR 381
IWRITE 383
IXTRIG 384
_SICLCLEANUP 387

A. The SICL Files

B. Updating HP-UX 9 SICL Applications

Building SICL Applications on HP-UX 11i 399

Linking with the Archive
Library on HP-UX 9 400

C. The SICL Utilities

iclear 403
ipeek 405
ipoke 406
iread 407
iwrite 408

D. Customizing your VXI/MXI System

Overview of VXI/MXI Configuration 411
The VXI/MXI Resource Manager (ivxirm) 412
The VXI/MXI Configuration Files 413
The vximanuf.cf Configuration File 414
The vximodel.cf Configuration File 414
The dynamic.cf Configuration File 414
The vmedev.cf Configuration File 415
The irq.cf Configuration File 415
The cmdrsrvt.cf Configuration File 416
The names.cf Configuration File 416
The oride.cf Configuration File 416
The ttltrig.cf Configuration File 417
The iproc Utility (Initialization and SYSRESET) 418
Viewing the VXIbus System Configuration 421
VXI/MXI Configuration Utilities 422
iproc 423
ivxirm 425
ivxisc 428

1

Introduction

Introduction

Welcome to the *Standard Instrument Control Library (SICL) User's Guide*. This manual describes how to use SICL. A getting started chapter steps you through the process of building and running a simple SICL program. The basics of SICL programming are covered in the following chapter, and later chapters describe how to use SICL with specific interfaces; GPIB, GPIO, VXI/MXI, RS-232, and, LAN. Also included is a complete SICL language Reference.

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

This manual contains the following:

- **Chapter 2 - Getting Started with SICL** steps you through building and running a simple example program. This is a good place to start if you are a first-time SICL user.
- **Chapter 3 - Using SICL** describes the basics of SICL along with some detailed example programs. You can find information on communication sessions, addressing, error handling, and more.
- **Chapter 4 - Using SICL with GPIB** describes communicating over the GPIB interface. Example programs are also provided.
- **Chapter 5 - Using SICL with GPIO** describes how to communicate over the GPIO interface. Example programs are also provided.
- **Chapter 6 - Using SICL with VXI/MXI** provides detailed information about communicating over the VXIbus.
- **Chapter 7 - Using SICL with RS-232** describes how to communicate over the RS-232 interface. Example programs are also provided.
- **Chapter 8 - Using SICL with LAN** describes how to communicate over a LAN. Example programs are also provided.
- **Chapter 9 - Troubleshooting Your SICL Program** describes some of the most common SICL programming problems and provides hints to help you solve the problems.
- **Chapter 10 - SICL Language Reference** provides a complete description of all of the available SICL functions and C languag syntax.

This guide also contains the following appendices:

- **Appendix A - The SICL Files** summarizes where the SICL files are located on your system.
- **Appendix B - Updating HP-UX 9 SICL Applications** describes how to update your SICL applications that were written on HP-UX 9 to work on HP-UX 11i.
- **Appendix C - The SICL Utilities** describes the SICL utilities that can be used to read and write to devices or interfaces from the command line.
- **Appendix D - Customizing your VXI/MXI System** documents how you can customize your VXI/MXI system. VXI/MXI configuration utilities are documented as well.

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

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 or C++ 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 the programmer complete control over I/O communications.

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
- Asynchronous event notification

SICL User

SICL is intended for instrument I/O and C/C++ programmers who are familiar with the HP-UX or Linux operating system. This manual does not attempt to teach the C programming language or instrument I/O concepts.

Related Documents

Other SICL Learning Products

- *I/O Libraries Installation and Configuration Guide* provides a detailed installation procedure with information on how to configure your system to run SICL.
- *SICL Online Help* is provided in the form of Unix manual pages (man pages).
- *SICL Example Programs* are provided in the `/opt/sicl/share/examples` directory. These examples are designed to help you develop your SICL applications more easily.

Other Documentation

- HP-UX 11i Learning Products (http://docs.hp.com/)
  - *HP-UX 11i Installation and Update Guide*
  - *HP C/HP-UX Reference Manual*
  - *HP-UX Linker and Libraries User's Guide*
  - *Software Distributor Administration Guide for HP-UX 11i*
  - *Managing Systems and Workgroups: A Guide for HP-UX System Administrators*

- Linux Learning Products
  - *Redhat Linux Installation Guide*
  - GCC Manuals (http://www.gnu.org/)
  - *Linux Network Administrator's Guide* by Olaf Kirch (O'Reilly & Associates)

- VXI Interface Learning Products
  - *TAMS 80100B PCI-VXI Controller Installation & Operations Instructions*

- GPIO Interface Learning Products
  - *TAMS PCI GPIO Card (71622/81622) Installation and Operations Instructions*

- GPIB Interface Learning Products
  - *TAMS PCI GPIB Card (70488/80488) Installation and Operations Instructions.*
  - *HP/Agilent E2078A User's Guide.*
  - *Tutorial Description of the Hewlett-Packard Interface Bus (HPIB)*

- Series 700 RS-232 Interface Learning Products
  - *The RS-232 Solution by Joe Campbell, SYBEX Publishing*

- LAN Learning Products
  - *Networking Overview*
  - *Installing and Administering LAN/9000 Software*
  - *Administering ARPA Services*

- LAN/GPIB Gateway Learning Products
  - *TAMS 3010 LAN I/O Gateway Installation and Configuration Guide*
  - *HP/Agilent E2050 LAN/GPIB Gateway Installation and Configuration Guide*

- VXIbus Consortium Specifications
  - *The VMEbus Specification*
  - *The VMEbus Extensions for Instrumentation*
  - *TCP/IP Instrument Protocol Specification - VXI-11, Rev. 1.0*
  - *TCP/IP-VXIbus Interface Specification - VXI-11.1, Rev. 1.0*
  - *TCP/IP-IEEE 488.1 Interface Specification - VXI-11.2, Rev. 1.0*
  - *TCP/IP-IEEE 488.2 Instrument Interface Specification - VXI-11.3, Rev. 1.0*

2

Getting Started with SICL

Getting Started with SICL

This chapter steps you through building and running your first SICL program. If you plan to develop SICL applications, go through this chapter to ensure you perform all the steps required to build and run a SICL program.

This chapter contains the following sections:

- Reviewing an SICL Program
- Compiling and Linking an SICL Program
- Running an SICL Program
- Getting Online Help
- Where to Go Next

If you need additional information on any of the SICL functions, see Chapter 10 for details.

Reviewing a SICL Program

Example programs are included in your SICL product to help you get started using SICL. Copies of the example programs are located in the `/opt/sicl/share/examples` directory.

The following is a simple C program that uses SCPI commands to query an GPIB instrument for its identification string and print the results.

```
/* idn.c
     The following program uses SICL to query an HPIB
instrument for an identification string and prints the
results. */
#include <stdio.h>
#include <sicl.h>              /* SICL header file */

/* Modify this line to reflect the address of your device */
#define DEVICE_ADDRESS "hpib,0"
void main()
{
   /* declare a device session id */
   INST id;
   char buf[256];

   /* error handler to exit if an error is detected */
   ionerror(I_ERROR_EXIT)

   /* open a device session with device at DEVICE_ADDRESS */
   id = iopen (DEVICE_ADDRESS);

   /* set timeout value to 1 sec */
   itimeout (id, 1000);

   /*send SCPI *RST command and prompt for id string*/
   iprintf (id, "*RST\n");
   ipromptf (id, "*IDN?\n", "%t", buf);

   /* print contents of buf */
   printf ("%s\n", buf);

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

Note The newline character (`\n`) in the `iprintf` and `ipromptf` functions in the previous example flushes the output buffer to the device and appends an END indicator to the newline. Sometimes flushing is needed for the device, and it is good practice to include this after each instrument command. You can specify when the buffer is flushed with the SICL `isetbuf` function. See Chapter 10 for information on this SICL function.

The SICL example program includes the following:

sicl.h This header file must be included at the beginning of your program to provide the function prototypes and constants defined by SICL.

DEVICE_ADDRESS This constant is defined specifically for this example. It is used to specify the device address. This address is then used in the `iopen` function call.

INST This is a type definition defined by SICL. It is used to represent a unique identifier that describes a specific device or interface.

ionerror This is a SICL function that installs an error handler that is automatically called if any SICL calls result in an error. `I_ERROR_EXIT` specifies that the error message is printed out and the program exited.

iopen This SICL function creates a device session with the device attached to the address specified in `DEVICE_ADDRESS` constant.

itimeout This function is called to set the length of time that SICL will wait for an instrument to respond. Different timeout values can be set for different sessions as needed.

iprintf, ipromptf These formatted I/O functions are patterned after those used in the C programming language. They support the standard ANSI C format string, plus added formats defined specifically for instrument I/O.

iclose This function closes the session with the specified device.

For more details on SICL features, see Chapter 3, "Using SICL." You can also see Chapter 10 for specifics about these SICL function calls.

Compiling and Linking a SICL Program

You can create your SICL applications in C, ANSI C, or C++. When compiling and linking a C program that uses SICL, use the `-lsicl` command line option to link in the SICL library routines. The following example creates the `idn` executable file on HP-UX 11i:

```
cc -Aa -o idn idn.c -lsicl
```

on Linux, use

```
gcc -o idn idn.c -lsicl
```

- The -Aa option specifies ANSI C on HP-UX
- The `-o` option creates an executable file called `idn`.
- The `-l` option links in the shared SICL library.

If you are building an application that was originally built on HP-UX 9, or if you need to link with the SICL archive libraries on HP-UX 9, see Appendix B, "Updating HP-UX 9 SICL Applications."

Using Shared Libraries

If your program uses a shared library that calls SICL, you must explicitly link the SICL library routines even if your program does not call SICL functions. If any part of your program performs instrument I/O, you must link the SICL library routines.

The following example shows the process of creating a shared library that calls SICL and using it with an end program on HP-UX 11i:

```
cc -Aa +z -c library.c -lsicl
ld -b -o library.sl library.o
cc -Aa -o y y.c library.sl -lsicl
```

on Linux, use

```
gcc -c library.c -lsicl
ld -shared -o library.so library.o
gcc -o y y.c -L. -llibrary -lsicl
```

Note If you fail to link the SICL library routines, you may get duplicate symbol errors when linking the end program or you may get undefined symbol errors memory fault (coredump) errors when you run the program.

Running an SICL Program

Execute your SICL program by typing the program name at the command prompt. For example:

```
idn
```

When using an HP/Agilent 54601A Four Channel Oscilloscope, you should get something similar to the following:

```
Hewlett-Packard,54601A,0,1.7
```

If you have problems running the `idn` example program, first check to make sure the device address specified in your program is correct. If the program still doesn't run, check the I/O configuration by running the `iosetup` utility. See the *I/O Libraries Installation and Configuration Guide* for information on running `iosetup`.

Getting Online Help

Online help is offered in the form of Unix manual pages (`man` pages). You can get help on the following SICL functions:

- SICL function calls
- SICL utilities

Using Manual Pages

To use manual pages, type the Unix `man` command followed by the SICL function call or utility:

```
man name
```

The following are examples of getting online help on SICL function calls and utilities: Examples of SICL function calls:

```
man iprintf
man ipromptf
man iread
```

Examples of SICL utilities:

```
man ipeek
man iread
man ivxisc
```

Where to Go Next

Once you have your SICL example program running, you can continue with Chapter 3, "Using SICL." Additionally, you should look at the chapters that describe how to use SICL with your particular I/O interface:

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

If you have any problems, see Chapter 9, "Troubleshooting Your SICL Program."

3

Using SICL

Using SICL

This chapter first describes how to use SICL and some of the basic features, such as error handling and locking. Detailed example programs are also provided to help you understand how these features work. Copies of the example programs are located in the `/opt/sicl/share/examples` directory.

This chapter contains the following sections:

- Including the `sicl.h` Header File
- Opening a Communications Session
- Sending I/O Commands
- Using Asynchronous Events
- Using Error Handlers
- Using Locking

For specific details on SICL function calls, see Chapter 10.

Including the sicl.h Header File

You must include the `sicl.h` header file at the beginning of every file that contains SICL calls. This header file contains the SICL function prototypes and the definitions for all SICL constants and error codes:

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

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 specific device connected to 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 commander. Typically a commander session is used when a computer connected to the interface is acting like a device.

There are two parts to opening a communication session with a specific device, interface, or commander. First, you must create an instance of a SICL session by declaring a variable of type `INST`. Once the variable is declared, then you can open the communication channel by using the SICL `iopen` function:

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

Where *id* is declared with the type `INST` and communicates to a device, interface, or commander. The *addr* parameter is a string expression which specifies a device session address, interface session address, or a commander session address. 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 `INST` 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 of programming, best overall performance, and best portability.

Addressing Device Sessions

To create a device session, specify either the interface `symbolic name` or `logical unit` and a particular device's address in the *addr* parameter of the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

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

Note Secondary addressing is *not* supported on the VXI and RS-232 interfaces.

The following are valid device addresses:

`7,23`	Device at bus address 23 connected to an interface card at logical unit 7.
`7,23,1`	Device at bus address 23, secondary address 1, connected to an interface card at logical unit 7.
`hpib,23`	Device at bus address 23 and symbolic name hpib.
`hpib2,23,1`	Device at bus address 23, secondary address 1, connected to a second GPIB interface with symbolic name hpib2.
`vxi,128`	Device at logical address 128 and symbolic name vxi.

The following is an example of opening a device session with the GPIB device at bus address 23:

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

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

Interface Sessions

An interface session allows 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, VXI, etc). 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 either the interface `symbolic name` or `logical unit` in the *addr* parameter of the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

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

The following are valid interface addresses:

`7`	Interface card at logical unit 7.
`hpib`	GPIB interface with the symbolic name hpib.
`hpib2`	Second GPIB interface with the symbolic name hpib2.

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

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

More on addressing specific interfaces can be found in the interface-specific chapter (for example, "Using SICL with GPIB") 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. However, when the controller is no longer the active controller, or passes control, commander sessions can be used to talk to the controller. In this mode, the interface is acting like a device on the interface (non-controller).

Addressing Commander Sessions

To create a commander session, specify either the interface `symbolic name` or `logical unit` followed by a comma and then the string `cmdr` in the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values. The following are valid commander addresses:

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

The following is an example of creating a commander session with the GPIB interface:

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

Sending I/O Commands

Once you have established a communications session with a device, interface, or commander, you can start communicating with that session using either formatted I/O or non-formatted I/O.

- 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 are very efficient in I/O.

- Non-formatted I/O sends or receives raw data to or from a device, interface, or commander. With non-formatted I/O, no formatting 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 and non-formatted I/O.

Formatted I/O

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 are as follows:

- The `iprintf` function formats according to the *format* string and sends data to the session specified by *id*:

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

- The `iscanf` function receives data from the session specified by *id* and converts the data according to the *format* string:

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

- The `ipromptf` function formats data according to the *writefmt* string and sends data to the session specified by *id* and then immediately receives the data and converts it according to the *readfmt* string:

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

See Chapter 10 for more information on these functions.

The formatted I/O functions are buffered. There are two non-buffered and non-formatted I/O functions called `iread` and `iwrite`. See the "Non-formatted I/O" section 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 data to or from the formatted I/O buffers. Refer to the "Formatted I/O Buffers" section later in this chapter 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 each 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 Chapter 10 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`

Flag	Description
@1	Converts to a IEEE 488.2 NR1 number.
@2	Converts to a IEEE 488.2 NR2 number.
@3	Converts to a IEEE 488.2 NR3 number.
@H	Converts to a IEEE 488.2 hexadecimal number.
@Q	Converts to a IEEE 488.2 octal number.
@B	Converts to a IEEE 488.2 binary number.
+	Prefixes number with sign (+ or -).
–	Left justifies result.
space	Prefixes number with blank space if positive or with - if negative.
#	Use alternate form. For o conversion, print 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 the left pad character to be a zero for all numeric conversion types.

The following example converts `numb` into a IEEE 488.2 floating point number (NR2) and sends it to the session specified by `id`:

```
int numb = 61; iprintf (id, "%@2d", numb);
Sends:  61.000000
```

Field Width. Field width is an optional integer that specifies the minimum number of characters 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`:

```
int numb = 61;
iprintf (id, "%6d", numb);
```

Inserts four characters, for a total of six characters: `  61`

.Precision. Precision is an optional integer that is preceded by a period. When used with conversion characters `e`, E, and `f`, the number of digits to the right of the decimal point is 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", 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 argument. 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", list);
```

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

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

Argument Modifiers

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 an array 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 an array of short integers.
z	b, B	Corresponding argument is pointer to an array of floats.
Z	b, B	Corresponding argument is a pointer to an array 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**

Conversion Character	Description
d, i	Corresponding argument is an integer
f	Corresponding argument is a double.
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 is 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:

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

Sends 1024 characters of block data.

`iscanf` **and** `ipromptf` **Conversion Characters**

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 sequence.
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.

The following example reads characters up to the first white space character from the session specified by the `id` parameter and puts the characters into data:

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

Formatted I/O Example

The following ANSI C example shows how to use the formatted I/O functions to send and receive data. This example opens an GPIB communications session with a Multimeter and sends a comma operator to send a comma separated list to the Multimeter. The `lf` conversion characters are then used to receive a double back 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};
   char buf[80];

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

   /* Open the multimeter session */
   dvm = iopen ("hpib,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);
}
```

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. All characters in the output buffer will be written 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. If no newline character is included in the format string for an `iprintf` call, then the converted characters are stored in the output buffer. It will require another call to `iprintf` or a call to `iflush` to have those characters written. `iflush` only sends the data queued in the buffer, and not the END indicator as in `iprintf`. Note that newline characters output from an output parameter do not cause a flush; only newlines in the format string do.

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 next section, "Formatted I/O Buffers."

The format string for `iscanf` ignores most white-space characters. Newlines (`\n`) and carriage returns (`\r`), however, are treated just like normal characters in the format string, which *must* match the next non-white-space character read.

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 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.

The read buffer is maintained by the `iscanf` and the read portion of the `ipromptf` functions. It queues the data received 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 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.

Overview of Formatted I/O

The following set of functions 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, 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. The advantage of this function is that the iprintf and iscanf parts are done together.
`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 in using buffers that are automatically allocated.

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 or write raw data to or from 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:

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

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

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

See Chapter 10 for more information on these functions.

Non-formatted I/O Example

The following example illustrates using non-formatted I/O to communicate with a Multimeter over the GPIB interface The SICL non-formatted I/O functions `iwrite` and `iread` are used for the communication. A similar example is used to illustrate formatted I/O later in this chapter.

```
/* nonformatio.c
   This example program measures AC voltage on a
   multimeter and prints out the results */
#include <sicl.h>
#include <stdio.h>

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

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

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

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

   /* Set up multimeter and take measurement */
   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, NULL);

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

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

}
```

Using Asynchronous Events

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. Any device can generate an SRQ. Both devices and interfaces can generate interrupts.

By default, creating a session enables asynchronous events. However, the library will not report any events to the application until the appropriate handlers are installed in your program.

SRQ Handlers

The `ionsrq` function installs an SRQ handler. The currently installed SRQ handler is called any time its corresponding device or interface 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, and have handlers installed, 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 notification of 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 Chapter 10.

Default is `on`.

Note It is possible to overflow SICL's interrupt queue if too many interrupts are generated while notification is disabled.

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 notification of 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 before enabling asynchronous events (that is, use `iintroff`). The `iwaithdlr` function behaves as if interrupts are enabled. Interrupts are still disabled after the `iwaithdlr` function has completed. Only calls to `iintron` will re-enable interrupts.

Note Interrupts must be disabled if you are using `iwaithdlr`. Use `iintroff` to disable notification of interrupts.

The reason for disabling notification of interrupts is that the interrupt may occur between the `isetintr` and `iwaithdlr` and, if you only expect one interrupt, it might come before the `iwaithdlr`. Notification may not occur, that is, the handler may not get called. This may or may not be the effect you desire.

For example:

```
...
iintroff ()
ionintr (vxi, trigger_handler);
isetintr (vxi, I_INTR_TRIG, I_TRIG_TTL0 | I_TRIG_TTL7);
...
ivxitrigon (vxi, I_TRIG_TTL0);
while (!done)
   iwaithdlr (0);
iintron ();
...
```

Asynchronous Events and Unix Signals

Note If you are using SICL LAN, see the "LAN and Signal Handling" section in Chapter 8, "Using SICL with LAN."

SICL `hpib` and `vxi` interfaces use an Unix signal to implement interrupts and SRQs. The default SICL signal is `SIGUSR2`. This signal is managed completely by the SICL library. Your application must avoid SICL's signal completely. Do not attempt to mask it, send it, or install a handler for it.

If your application needs `SIGUSR2` for some purpose other than SICL, you can instruct SICL to use a different signal. This is done with the `isetsig` function. The following example selects signal 29 for SICL use:

```
isetsig(29);
```

If you use `isetsig`, you *must* call it before any other function in your program. Also, you must pick an alternate signal carefully to avoid conflicting with other Unix resources.

Protecting I/O Calls Against Interrupts

It is standard Unix behavior for I/O calls like `iread` and `iprintf` to be interrupted when the process receives a signal. If your process is not expecting to receive signals, such I/O side effects will probably be masked by the other standard behavior of unexpected signals: death of your process. If you are expecting signals, you may not want them to abort SICL I/O operations.

This can be solved by blocking or ignoring any expected signals while doing I/O activity. After I/O is complete, the original signal action can be restored. The choice to block or ignore depends on the need of your application. Ignored signals are not queued; blocked signals have a one-deep queue and are acted on as soon as the block is removed.

The following programming segment shows signal blocking. `SIGALARM` and `SIGINT` are blocked during an `iscanf` call.

```
.
.
/* temporarily block 2 signals */
 old_mask = sigblock(sigmask (SIGINT) | sigmask (SIGALRM));

/* call protected I/O function */
iscanf (id, "%f", &mydata);

/* restore original signal mask */
sigsetmask (old_mask);
```

Interrupt Handler Example

The following is an ANSI C example that installs an interrupt handler and enables the interrupts on the VXI TTL trigger lines. When the TTL trigger line is asserted, the installed interrupt handler is called.

```
/* interrupts.c
   * This is an example of the interrupt handling in SICL.  This
   * program installs an interrupt handler and enables the
   * interrupts on trigger and waits for the interrupt. */
#include <sicl.h>
#include <stdio.h>
#include <unistd.h>

int intr = 0;

void trigger_handler (INST id, long reason, long secval) {
   /* indicate that the interrupt happened */
   intr = 1;
}  /* end of trigger_handler */

main ()
{
   INST id;

   /* start child process to fire trigger line */
   if (fork()==0)
      child();

   ionerror (I_ERROR_EXIT);
   iintroff();

   id = iopen ("vxi");

   /* set the interrupt handler */
   ionintr (id, trigger_handler);

/* what interrupts to handle (interrupt on ttl 0 or 7 firing) */
   isetintr (id, I_INTR_TRIG, I_TRIG_TTL0 | I_TRIG_TTL7);
```

```
  /* Wait for interrupt to happen (30 second timeout) */
  iwaithdlr (30000);

  if (intr == 1)
    printf ("Interrupt handler called.\n");
  else
    printf ("ERROR:  Interrupt handler not called.\n");

  iclose (id);
}

child ()
{
  INST id;

  /* Let the parent get into iwaithdlr */
  sleep (2);

  ionerror (I_ERROR_EXIT);

  id = iopen ("vxi");

  /* pulse TTL0 */
  ivxitrigon (id, I_TRIG_TTL0);
  ivxitrigoff (id, I_TRIG_TTL0);

  iclose (id);
  exit (0);
}
```

Using Error Handlers

When a SICL function call 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 an application.

It is important to note that error handlers are per-process, *not* per-session. That is, one handler will work for all sessions in a process. 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.

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

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

Where:

```
void 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 print a diagnostic message and then terminate the process.
`I_ERROR_NO_EXIT`	This value installs a special error handler which will print 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.

Error Handler Example

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 a SICL call results in an error.

In this example a standard, system-defined error handler is installed that prints 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;

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

   exit (0);
}
```

The following is an ANSI C 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>

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

main () {
   INST dvm;
   double res;

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

   exit (0);
}
```

Now, if any of the SICL functions result in an error, your error routine will be called.

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

Using Locking

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. If a session within a given process locks a device or interface, then that device or interface can only be accessed from that session.

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 an interface which is serving a device locked from a device session. This interface access usually allows the interface session to address or reset any device on the interface. In such a case, data may be lost from the device session that was underway.

In particular, be aware that both the HP/Agilent Visual Engineering Environment (HP/Agilent VEE) and the TAMS BASIC applications use SICL interface sessions. Hence, I/O operations from either of these applications can supersede any device session that has a lock on a particular device. Use interface session locks in your own program if these applications may be running simultaneously with your program.

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. Each function defined in Chapter 10 has a section, "Affected by functions," that lists the keyword `LOCK` if the function is affected by locks. Functions without this keyword are not affected.

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 Chapter 10 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, or 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 help ensure exclusive use of a particular device or set of devices. (However, as explained in the previous section, "Using Locking," 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 Example

The following example show how device locking can be used to grant exclusive access to a device by an application. This example uses an HP/ Agilent 34401 Multimeter.

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

#include <sicl.h>
#include <stdio.h>
main() {
   INST dvm;

   char strres[20];

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

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

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

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

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

4

Using SICL with GPIB

Using SICL with GPIB

The HPIB interface (Hewlett-Packard Interface Bus) is Hewlett-Packard's implementation of the IEEE 488.1 Bus. Other IEEE 488 versions include GPIB (General Purpose Interface Bus) and IEEE Bus. GPIB and HPIB are both used in the discussions and examples in this chapter. The HPIB related SICL functions have the string GPIB embedded in the function name.

This chapter explains how to use SICL to communicate over GPIB. In order to communicate over GPIB, you must have loaded the GPIB fileset during the system installation. See the *I/O Libraries Installation and Configuration Guide* for information.

This chapter describes in detail how to open a communications session and communicate with GPIB devices, interfaces, or controllers. The example programs shown in this chapter are also provided in the `/opt/sicl/share/examples` directory.

This chapter contains the following sections:

- Creating a Communications Session with GPIB
- Communicating with GPIB Devices
- Communicating with GPIB Interfaces
- Communicating with GPIB Commanders
- Summary of GPIB Specific Functions

Creating a Communications Session with GPIB

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

Communicating with GPIB Devices

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 GPIB Devices

To create a device session, specify either the interface `symbolic name` or `logical unit` and a particular device's address in the *addr* parameter of the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

The following are example GPIB addresses for device sessions:

`hpib,7`	A device address corresponding to the device at primary address 7 and symbolic name `hpib`.
`hpib,3,2`	A device address corresponding to the device at primary address 3, secondary address 2, and symbolic name `hpib`.
`hpib,9,0`	A device address corresponding to the device at primary address 9, secondary address 0, and symbolic name `hpib`.

Note The above examples use the default `symbolic name` specified during the system configuration. If you want to change the name listed above, you must also change the `symbolic name` or `logical unit` specified during the configuration. The name used in your SICL program must match the `logical unit` or `symbolic name` specified in the system configuration. Other possible interface names are `GPIB`, `gpib`, `HPIB`, etc.

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 GPIB primary and secondary addresses.

Note If you are using an GPIB Command Module to communicate with VXI devices, the secondary address must be specified to select a specific instrument in the cardcage. Secondary addresses of 0, 1, 2, . . .31 correspond to VXI instruments at logical addresses of 0, 8, 16, . . . 248, respectively.

The following is an example of opening a device session with an GPIB device at bus address 16:

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

SICL Function Support with GPIB Device Sessions

The following describes how some SICL functions are implemented for GPIB device sessions.

`iwrite`	Causes all devices to untalk and unlisten. It then 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 device clear (DCL) on the device corresponding to this session.

GPIB Device Session Interrupts

There are no device-specific interrupts for the GPIB interface.

GPIB Device Sessions and Service Requests

GPIB device sessions support Service Requests (SRQ). On the GPIB interface, when one device issues an SRQ, the library will inform *all* GPIB device sessions that have SRQ handlers installed. (See `ionsrq` in Chapter 10.) This is an artifact of how GPIB handles the SRQ line. The interface cannot distinguish which device requested service. Therefore, the library acts as if all devices require service. Your SRQ handler can retrieve the device's **status byte** by using the `ireadstb` function. It is good practice to ensure that a device isn't requesting service before leaving the SRQ handler. The easiest technique for this is to service all devices from one handler.

The data transfer functions work only when the GPIB interface is the Active Controller. Passing control to another GPIB device causes the interface to lose active control.

GPIB Device Session Example

The following example illustrates communicating with an GPIB device session. This example opens two GPIB 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.

```
/* 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;

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

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

Communicating with GPIB Interfaces

Interface sessions allow you direct low-level control of the interface. You must do all the bus maintenance for the interface. This also implies that you have considerable knowledge of 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 GPIB Interfaces

To create an interface session on your GPIB system, specify either the interface `symbolic name` or `logical unit` in the *addr* parameter of the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

The following are example GPIB interface addresses:

`hpib`	An interface symbolic name.
`hpib2`	An interface symbolic name.
`7`	An interface logical unit.

Note The above examples use the default `symbolic name` specified during the system configuration. If you want to change the name listed above, you must also change the `symbolic name` or `logical unit` specified during the configuration. The name used in your SICL program must match the `logical unit` or `symbolic name` specified in the system configuration. Other possible interface names are `GPIB`, `gpib`, `HPIB`, `IEEE488`, etc.

The following example opens a interface session with the GPIB interface:

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

SICL Function Support with GPIB Interface Sessions

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

`iwrite`	Sends the specified bytes directly to the interface without performing any bus addressing. The `iwrite` function always clears the ATN line before sending any bytes, thus ensuring that the GPIB interface sends the bytes as data, not command bytes.
`iread`	Reads the data directly from the interface without performing any bus addressing.
`itrigger`	Performs a GPIB group execute trigger (GET) without additional addressing. This function should be used with the `igpibsendcmd` to send an `UNL` followed by the 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` routine 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 and REN), which resets the interface.

GPIB Interface Session Interrupts

There are specific interface session interrupts that can be used. See `isetintr` in Chapter 10 for information on the interface session interrupts.

There are no device specific interrupts for the GPIB interface.

GPIB Interface Sessions and Service Requests

GPIB interface sessions support Service Requests (SRQ). On the GPIB interface, when one device issues an SRQ, the library will inform *all* GPIB interface sessions that have SRQ handlers installed. (See `ionsrq` in Chapter 10.) It is good practice to ensure that a device isn't requesting service before leaving the SRQ handler. The easiest technique for this is to service all devices from one handler.

GPIB Interface Session Examples

Checking the Bus Status

The following example program is an ANSI C program that retrieves the GPIB interface bus status information and displays it for the user.

```
/* hpibstatus.c
   The following example retrieves and displays HPIB bus
    status information. */
#include <stdio.h>
#include <sicl.h>

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

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

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

   /* retrieve HPIB 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);
   return 0;
}
```

Communicating with Devices via Interface Sessions

The following example program sets up two GPIB instruments over an interface session and has the instruments communicate with each other.

The 3 main parts of this program are as follows:

- Read the data from the scope (`get_data`).
- Print some statistics about the data (`massage_data`).
- Have the scope send the data to a printer (`print_data`).

```
/* hpibintr.c
   This program requires a 54601A digitizing oscilloscope
   or compatible) and a printer capable of printing in HP
   RASTER GRAPHICS STANDARD (e.g. thinkjet).
   This program will tell the scope to take a reading on
   channel 1, then send the data back to this program.
   Then  some simple statistics about the data is printed.
   The program then tells the scope to send the data
   directly to the printer, illustrating how the
   controller does not have to be directly involved in an
   HPIB transaction.*/

#include <stdio.h>   /* used for printf() */
#include <stdlib.h>  /* used for exit() */
#include <sicl.h>    /* SICL header file */

/* defines */
#define INTF_ADDR    "hpib"
#define SCOPE_ADDR   INTF_ADDR ",7"

/* function prototypes */
void initialize (void);
void get_data (void);
void massage_data (void);
void print_data (void);
void cleanup (void);
void srq_hdlr (INST id);

/* global data */
float pre[10];
INST scope;
INST intf;
```

```
void main() {
   ionerror(I_ERROR_EXIT);
   scope = iopen(SCOPE_ADDR);
   intf = iopen(INTF_ADDR);

   initialize();
   get_data();
   massage_data();
   print_data();
   cleanup();

   iclose(scope);
   iclose(intf);
}

void initialize() {
   /* initialize the hpib interface and scope */
   iclear(intf);
   itimeout(scope, 5000);
   itimeout(intf, 5000);
   iclear(scope);
   igpibllo(intf);
}

void get_data() {
   short readings[5000];
   int count;

   /* setup scope to accept waveform data */
   iprintf(scope, "*RST\n");
   iprintf(scope, ":autoscale\n");

   /* setup up the waveform source */
   iprintf(scope, ":waveform:format word\n");

   /* input waveform preamble to controller */
   iprintf(scope, ":digitize channel1\n");
   iprintf(scope, ":waveform:preamble?\n");
   iscanf(scope, "%,10f", pre);

   /* command scope to send data */
   iprintf(scope, ":waveform:data?\n");
```

```
   /* enter the data */
   count = 5000;
   iscanf(scope, "%#wb\n", &count, readings);
   printf ("received %d words\n", count);
}

void massage_data() {
   float vdiv;
   float off;
   float sdiv;
   float delay;
   char  id_str[50];

   vdiv  = 32 * pre[7];
   off   = (128 - pre[9]) * pre[7] + pre[8];
   sdiv  = pre[2] * pre[4] / 10;
   delay = (pre[2] / 2 - pre[6]) * pre[4] + pre[5];

   /* retrieve the scope's ID string */
   ipromptf(scope, "*IDN?\n", "%s", id_str);

    /*  print the statistics about the data */
   printf("\nOscilloscope ID:  %s\n", id_str);
   printf(" ----------  Current settings  -----------\n");
   printf("      Volts/Div = %f V\n", vdiv);
   printf("         Offset = %f V\n", off);
   printf("          S/Div = %f S\n", sdiv);
   printf("          Delay = %f S\n", delay);
}

void print_data() {
   unsigned char status;
   char    cmd[5];

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

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

```
   /* 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] = 63;    /* 63 is unlisten                          */
cmd[1] = 32+1; /* printer is at address 1, make it a listener*/
cmd[2] = 64+7; /* scope is at address 7, make it a talker*/
cmd[3] = '\0';  /* terminate the string                    */

igpibsendcmd(intf, cmd, 3);

   /* set up our SRQ handler to be called when the scope
      finishes printing */
   ionsrq(scope, srq_hdlr);

   /* now, the ATN line must be set to FALSE */
   igpibatnctl(intf, 0);

/* wait for SRQ before continuing program */
   status = 0;
   while(status == 0) {
      iwaithdlr(120000L);

      /* make sure it was the scope requesting service */
      ireadstb(scope, &status);
      status &= 64;
   }

   /* clear the status byte so the scope can assert SRQ
       again if needed. */
   iprintf(scope, "*CLS\n");
}

void cleanup() {
   /* give local control back to the scope */
   ilocal(scope); }

void srq_hdlr(INST id) {
   /* this handler does nothing. we will use iwaithdlr()in
the code above to determine when the handler gets called. */
}
```

Communicating with GPIB Commanders

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

Addressing GPIB Commanders

To create a commander session on your GPIB interface, specify either the interface `symbolic name` or `logical unit` in the *addr* parameter followed by a comma and the string `cmdr` in the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

The following are example GPIB addresses for commander sessions:

`hpib,cmdr`	A commander session with the `hpib` symbolic name.
`hpib2,cmdr`	A commander session with the `hpib2` symbolic name.
`7,cmdr`	A commander session with the interface at logical unit 7.

Note The above examples use the default `symbolic name` specified during the system configuration. If you want to change the name listed above, you must also change the `symbolic name` or `logical unit` specified during the configuration. The name used in your SICL program must match the `logical unit` or `symbolic name` specified in the system configuration. Other possible interface names are `GPIB`, `gpib`, `HPIB`, etc.

The following example opens a commander session the GPIB interface:

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

SICL Function Support with GPIB Commander Sessions

The following describes how some SICL functions are implemented for GPIB commander sessions.

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

GPIB Commander Session Interrupts

There are specific commander session interrupts that can be used. See `isetintr` in Chapter 10 for information on the commander session interrupts.

Summary of GPIB Specific Functions

Note Using these GPIB interface specific functions means that the program can not be used on other interfaces and, therefore, becomes less portable.

SICL GPIB Functions

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

5

Using SICL with GPIO

Using 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 GPIB, its speed and versatility make it the perfect choice for many tasks.

This chapter explains how to use SICL to communicate over GPIO. In order to communicate over GPIO, you must have loaded the GPIO fileset during the I/O Libraries installation. See the *I/O Libraries Installation and Configuration Guide* for information. Also note that the GPIO related SICL functions have the string `GPIO` embedded in their names.

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 `/opt/sicl/share/examples` directory.

Note GPIO is *not* supported with SICL over LAN.

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 GPIB and VXI, 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 either the interface `symbolic name` or `logical unit` in the *addr* parameter of the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

The following are example addresses for GPIO interface sessions:

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

Note The above examples use the default `symbolic name` specified during the system configuration. If you want to change the name listed above, you must also change the `symbolic name` or `logical unit` specified during the configuration. The name used in your SICL program must match the `logical unit` or `symbolic name` specified in the system configuration. Other possible interface names are `parallel`, `GPIO`, etc.

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

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

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 the Series 700, these bytes are ordered: high-low-high-low... Because of this "unpacking" operation, 16-bit data widths may not be appropriate for formatted I/O operations. For `iscanf` termination, an END value must be specified using `igpioctrl`. See Chapter 10 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 microsecond. 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 approximately 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 *mode* configuration specified during the SICL configuration.

`ionsrq`	Installs a service request handler for this session. The concept of service request (SRQ) originates from GPIB. On an GPIB interface, a device can request service from the controller by asserting a line on the interface bus. On GPIO, the EIR line is assumed to be the service request line.
`ireadstb`	Chapter 10 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 Chapter 10 for information on the interface session interrupts for GPIO.

GPIO Interface Session Example

```
/* gpiomeas.c
   This program does the following:
   - Creates 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 */

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

   /* open GPIO interface session, with 10-second timeout
*/
   id = iopen ("gpio");
   itimeout (id, 10000);

   /* 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);

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

GPIO Interrupts Example

```
/* gpiointr.c
   This program does the following:
   - Creates a GPIO session with error checking
   - Installs interrupt handler & enables EIR interrupts
   - Waits for EIR; invokes the handler for each interrupt
   - Handler checks interrupt cause & exits when EIR is
     clear
*/
#include <sicl.h>

void handler(id, reason, sec
INST id;
long 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 */

   /* 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);
    }

    /* 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 (intf);
}
```

Summary of GPIO Specific Functions

Note Using these GPIO interface specific functions means that the program can not be used on other interfaces and, therefore, becomes less portable.

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

Request	Characteristic	Settings
I_GPIO_AUTO_HDSK	Auto-Handshake mode	1 or 0
I_GPIO_AUX	Auxiliary Control lines	16-bit mask
I_GPIO_CHK_PSTS	Check PSTS before read/write	1 or 0
I_GPIO_CTRL	Control lines	I_GPIO_CTRL_CTL0 I_GPIO_CTRL_CTL1
I_GPIO_DATA	Data Output lines	8-bit or 16-bit mask
I_GPIO_PCTL_DELAY	PCTL delay time	0-7
I_GPIO_POLARITY	Logical polarity	0-31
I_GPIO_READ_CLK	Data input latching	See *Chapter 10*
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

6

Using SICL with VXI/MXI

Using SICL with VXI/MXI

This chapter explains how to use SICL to communicate over the VXIbus. In order to communicate directly over the VXIbus, you must have loaded the VXI fileset during the I/O Libraries installation. See the *I/O Libraries Installation and Configuration Guide* for information. The example programs shown in this chapter are also provided in the `/opt/sicl/share/examples` directory.

This chapter contains the following sections:

- Creating a Communications Session with VXI/MXI
- Communicating with VXI/MXI Devices
- Communicating with VME Devices
- Communicating with VXI/MXI Interfaces
- Looking at SICL Function Support with VXI/MXI
- Using SICL Trigger Lines
- Using `i?blockcopy` for DMA Transfers
- Using VXI Specific Interrupts
- Summary of VXI/MXI Specific Functions

For information on the specific SICL function calls, see Chapter 10.

Creating a Communications Session with VXI/MXI

Before you start programming your VXI/MXI system, ensure that the system is set up and operating correctly. See Appendix D, "Customizing your VXI/MXI System," later in this manual for configuration information.

To begin programming your VXI/MXI system, you must determine what type of communication session you need. The two supported VXI communication sessions are as follows:

Device Session	The device session allows you direct access to a device without worrying about the type of interface to which it is connected.
Interface Session	An interface session allows direct low-level control of the specified interface. This gives you full control of the activities on a given interface, such as VXI.

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

Note Commander Sessions are *not* supported with VXI interfaces.

Communicating with VXI/MXI Devices

If you are going to use SICL functions to communicate directly with VXI devices, you must first be aware of the two different types of VXI devices:

Message-Based — Message-based devices have their own processors which allow them to interpret the high-level SCPI (Standard Commands for Programmable Instruments) commands. While using SICL, you simply place the SCPI command within your SICL output function call, and the message-based device interprets the SCPI command.

Register-Based — The register-based device typically does not have a processor to interpret high-level commands; and therefore, only accepts binary data. Use the following methods to program register-based instruments:

- **Interpreted SCPI -** Use the SICL `iscpi` interface and program using high-level SCPI commands. I-SCPI interprets the high-level SCPI commands and sends the data to the instrument. Interpeted SCPI drivers for cards are available only for selected operating systems and cards. These drivers rely on card specific information and therefore are usually provided and distributed by the card manufacturer. At the time of writing, the instrument manufacturers had not ported any of their drivers to HP-UX 11i or Linux. For availability information, contact the manufacturer of your VXI device. You may also contact TAMS for more information.

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

Note Interpreted SCPI (I-SCPI) is supported over LAN. However, register programming (`imap`, `ipeek`, `ipoke`, etc) is *not* supported over LAN.

I-SCPI runs on the LAN server if used in a LAN-based system.

Other Products:

- HP/Agilent Command Module - Use a Command Module to interpret the high-level SCPI commands. The `hpib` interface is used with a Command Module. A Command Module may also be accessed over a LAN using a LAN-to-GPIB gateway, such as the HP/Agilent E2050 LAN to GPIB Gateway.

Programming with register-based and message-based devices is discussed in further detail later in this section.

Note You can program a VXIbus system that is mixed with both message-based and register-based devices. To do this, open a communications session for each device in your system and program as shown in the following sections.

Message-Based Devices

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

Note If your message-based device has shared memory, you can access the device's shared memory by doing register peeks and pokes. See "Register-Based Devices" later in this chapter for information on register programming.

Addressing VXI/MXI Message-Based Devices

To create a device session, specify either the interface `symbolic name` or `logical unit` and a particular device's address in the *addr* parameter of the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

The following are example addresses for VXI/MXI device sessions:

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

Remember that the primary address must be between 0 and 255. The primary address corresponds to the VXI logical address and specifies the address in A16 space of the VXI device.

Note The previous examples use the default `symbolic name` specified during the system configuration. If you want to change the name listed above, you must also change the `symbolic name` or `logical unit` specified during the configuration. The name used in your SICL program must match the `logical unit` or `symbolic name` specified in the system configuration. Other possible interface names are `VXI`, `MXI`, `mxi`, etc.

SICL supports only primary addressing on the VXI device sessions. Specifying a secondary address causes an error.

The following is an example of opening a device session with the VXI device at logical address 64:

```
INST dmm;
dmm = iopen ("vxi,64");
```

Message-Based Device Session Example

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

```
/* vximesdev.c
   This example program measures AC voltage on a
    multimeter and prints out the results */
#include <sicl.h>
#include <stdio.h>

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

   /* Print message and terminate on error */
   ionerror (I_ERROR_EXIT);
   /* Open the multimeter session */
   dvm = iopen ("vxi,24");
   itimeout (dvm, 10000);

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

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

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

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

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

}
```

Register-Based Devices

There are several methods that can be used for communicating with register-based devices:

`iscpi` interface	Use the SICL `iscpi` interface and program using SCPI commands. The `iscpi` interface interprets the SCPI commands and allows you to communicate directly with register-based devices.Interpeted SCPI drivers for cards are available only for selected operating systems and cards. These drivers rely on card specific information and therefore are usually provided and distributed by the card manufacturer. At the time of writing, the instrument manufacturers had not ported any of their drivers to HP-UX 11i or Linux. For availability information, contact the manufacturer of your VXI device. You may also contact TAMS for more information.
Register Programming	Use the `vxi` interface to program directly to the device's registers with a series of register peeks and pokes. This method can be very time consuming and difficult. This method is *not* supported over LAN.
Other Products	
HP/Agilent Command Module	HP/Agilent Command Module\When you use an HP/Agilent Command Module to communicate with VXI/MXI devices, you are actually communicating over GPIB. The command module interprets the high-level SCPI commands for register-based instruments and then sends out low-level commands over the VXIbus backplane to the instruments. See the "Using SICL with GPIB" chapter for more details on communicating through a command module.

Note There are also other applications that use SICL as their I/O library but have their own methods of communicating with the instruments. These applications hide most of the I/O complexity behind the user interface.

Addressing VXI/MXI Register-Based Devices

To create a device session, specify either the interface `symbolic name` or `logical unit` and a particular device's address in the *addr* parameter of the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

The following are example addresses for VXI/MXI device sessions:

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

Remember that the primary address must be between 0 and 255. The primary address corresponds to the VXI logical address.

Note The above examples use the default `symbolic name` specified during the system configuration. If you want to change the name listed above, you must also change the `symbolic name` or `logical unit` specified during the configuration. The name used in your SICL program must match the `logical unit` or `symbolic name` specified in the system configuration. Other possible interface names are `VXI`, `MXI`, `mxi`, etc.

SICL supports only primary addressing on the VXI device sessions. Specifying a secondary address causes an error.

The following is an example of opening a device session with the VXI device at logical address 64:

```
INST dmm;
dmm = iopen ("vxi,64");
```

Programming Directly to the Registers

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

When sending a series of peeks and pokes to the device's registers, use the following process:

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

Note Note that the above procedure is only used on register-based devices that are not using the `iscpi` interface.

Note that programming directly to the registers is not supported over LAN.

Mapping Memory Space for Register-Based Devices. When using SICL to communicate directly to the device's registers, you must map a memory space into your process space. This can be done by using the SICL `imap` function:

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

This function maps space for the interface or device specified by the *id* parameter. *pagestart*, *pagecnt*, and *suggested* are used to indicate the page number, how many pages, and a suggested starting location respectively. *map_space* determines which memory location to map the space. The following are valid *map_space* choices:

- `I_MAP_A16` Maps in VXI A16 address space (device or interface sessions, 64K byte pages).
- `I_MAP_A24` Maps in VXI A24 address space (device or interface sessions, 64K byte pages).
- `I_MAP_A32` Maps in VXI A32 address space (device or interface sessions, 64K byte pages).

- `I_MAP_VXIDEV` Maps in VXI A16 device registers (device session only, 64 bytes).
- `I_MAP_EXTEND` Maps in VXI device extended memory address space in A24 or A32 address space (device sessions only).
- `I_MAP_SHARED` Maps in VXI/MXI A24/A32 memory that is physically located on the computer (sometimes called local shared memory, interface sessions only).
- `I_MAP_AM` *address modifer* Maps in the specified region (*address modifer*) of VME address space. See the "Communicating with VME Devices," later in this chapter for more information on this map space argument.

The following are example `imap` function calls:

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

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

/* Map to A24 space while using E1489 (8 Mbytes) */
ptr = imap (id, I_MAP_A24, 0x00, 0x80, NULL);

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

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

Note Due to hardware constraints on given devices or interfaces, not all address spaces may be implemented. In addition, there may be a maximum number of pages that can be simultaneously mapped.

If a request is made that cannot be granted due to hardware constraints, the process will hang until the desired resources become available. To avoid this, use the `isetlockwait` with the *flag* parameter set to 0, and thus generate an error instead of waiting for the resources to become available. You may also use the `imapinfo` function to determine hardware constraints before making an `imap` call.

Reading and Writing to the Device Registers. Once you have mapped the memory space, use the SICL `i?peek` and `i?poke` functions to communicate with the register-based instruments. With these functions, you need to know which register you want to communicate with and the register's offset. See the instrument's user's manual for a description of the registers and register locations.

The following is an example of using iwpeek:

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

See Chapter 10 for a complete description of the `i?peek` and `i?poke` functions.

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

Register-Based Programming Example

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

Note The HP-UX Series 700 C++ compiler dereferences pointers that are cast to another data type by making multiple accesses of the base data type. Therefore, if you cast a character pointer to a short pointer, it will dereference it as two D08 accesses. To correct this problem, always use the size pointer that you would like the access to be. If you want D16 accesses, use a short pointer. If you want D32 accesses, use a long pointer. For example:

```
unsigned short *a24_ptr;

a24_ptr = (unsigned short *) imap (id, I_MAP_A24, ps, cnt, 0);
val = iwpeek (a24_ptr + offset);
```

```
/* vxiregdev.c
   The following example prompts the user for an
    instrument address and then reads the id register
    device type register.  The contents of the register
    are then displayed. */
#include <stdio.h>
#include <stdlib.h>
#include <sicl.h>

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

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

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

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

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

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

   /* print results */
   printf ("Instrument at address %s\n", inst_addr);
   printf ("ID Register = 0x%4X\n  Device Type Register = 0x%4X\n",
            id_reg, devtype_reg);

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

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

Catching Bus Errors Example

It is good practice to add bus error handling to your applications that use `i?peek` and `i?poke`. Add a `catch_buserror` function call before using `i?peek` or `i?poke` and the `uncatch_buserror` function call at the end of your application. The following is an example of these functions:

```
/* The following functions handle catching & processing
buserrors. */
#include <signal.h>

/* Structure defined in signal.h. */
struct sigaction oldact;

/* Handler called when there's a bus error.  It prints
    an error message and exits. */
static void be_handler (int)
{
   fprintf (stderr, "ERROR: Bus Error \n");
   exit (1);
}

/* Function to catch the bus error. */
void catch_buserror ()
{
   struct sigaction newact;

   /*Assign be_handler to be called when action is to be taken.*/
   newact.sa_handler = (void (*)(...)) be_handler;

   /* Assign SIGBUS as signal to be caught. */
   sigemptyset (&newact.sa_mask);
   sigaddset (&&newact.sa_mask, SIGBUS);

   /* Set sa_flags to 0. */
   newact.sa_flags=0;

   sigaction (SIGBUS, &newact, &oldact);
}

/* Function to release bus error. */
void uncatch_buserror()
{
   sigaction (SIGBUS, &oldact, 0);
}
```

Communicating with VXI/MXI Interfaces

Interface sessions allow you direct low-level control of the interface. You must do all the bus maintenance for the interface. This also implies that you have considerable knowledge of 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 VXI/MXI Interface Sessions

To create an interface session on your VXI/MXI system, specify either the interface `symbolic name` or `logical unit` in the *addr* parameter of the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

The following are example addresses for VXI/MXI interface sessions:

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

Note The above examples use the default `symbolic name` specified during the system configuration. If you want to change the name listed above, you must also change the `symbolic name` or `logical unit` specified during the configuration. The name used in your SICL program must match the `logical unit` or `symbolic name` specified in the system configuration. Other possible interface names are `VXI`, `MXI`, `mxi`, etc.

The following example opens a interface session with the VXI interface:

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

VXI/MXI Interface Session Example

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

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

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

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

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

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

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

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

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

Communicating with VME Devices

Note Not supported over LAN.

Many people assume that since VXI is an extension of VME that VME should be easy to use in a VXI system. Unfortunately, this is not true. Since the VXI standard defines specific functionality that would be a custom design in VME, some of the resources required for VME custom design are actually used by VXI. Therefore, there are certain limitation and requirements when using VME in a VXI system. Note that VME is not an officially supported interface for SICL.

Use the following process when using VME devices in a VXI/MXI mainframe:

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

Each of the above items are described in further detail in the following subsections. An example program is also provided.

Declaring Resources

The VXI Resource Manager does not reserve resources for VME devices. Instead, a configuration file is used to reserve resources for VME devices in a VXI system. Use the `/etc/opt/sicl/vxi`*LU*`/vmedev.cf` (where *LU* is the logical unit of the VXI/MXI interface) to reserve resources for VME devices. The VXI Resource Manager reads this file to reserve the VME address space and VME IRQ lines. The VXI Resource Manager then assigns the VXI devices around the already reserved VME resources.

When you edit the `vmedev.cf` file, you need to specify the device name, bus, slot #, address space, starting offset, size, and VME IRQ line. The following is an example entry:

```
vmedev1     0     12     A24     0x400000     0x10000     3
```

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

For VME devices using A24 or A32 address space, use A24 or A32 address ranges just higher than those used by your VXI devices. To determine what A24 or A32 address ranges are used by your VXI devices, run the Resource Manager (`ivxirm`) without the VME devices installed. Then edit the `vmedev.cf` file to specify the appropriate address range. This will prevent the Resource Manager (`ivxirm`) from assigning the address range used by the VME device to any VXI device. (The A24 and A32 address range is software programmable for VXI devices.)

E1482 VXI-MXI Resources

When a VME device is accessed via an E1482 VXI-MXI Extender Bus, you must declare the `bus` for a given VME device. The `bus` is declared as described in the previous section in the `vmedev.cf` file. For devices in a VXI/MXI system, use the logical address of the E1482 in the mainframe as the `bus`.

Additionally, since VME devices mapped in A16 address space are required to the use the lower 75% of A16 address space, the A16 Window Map Register of the E1482 must be programmed. To program this register, you must edit the `/etc/opt/sicl/vxi16/oride.cf` file to open an A16

address window for the device. An entry to this file changes the value SICL writes to the A16 window map register of the E1482.

The `oride.cf` file contains the logical address of the VXI-MXI Bus Extender card, the offset value, and the value written to the register. See the "Register Description" appendix of the E1482 user's manual for information on the value that should be placed in the `oride.cf` file. When using this appendix, it is important to note that SICL normally has the `CMODE` bit clear. The following example opens all of the lower 48k of A16 address space:

```
1  0xC  0x7800
```

Mapping VME Memory

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

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

Use the `I_MAP_AM` *address modifer* map space argument in the imap function to specify the map space region (*address modifer*) of VME address space. See the VMEbus Specifications for information on what value to use as the address modifier. Note that if the controller doesn't support specified address mode, then the `imap` call will fail (see table in the next section).

The following maps A24 non-privileged data access mode:

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

The following maps A32 non-privileged data access mode:

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

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

Supported Access Modes

The following tables list VME access modes supported on VXI controllers:

VME Controller Mapping Support

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

Reading and Writing to the Device Registers

Once you have mapped the memory space, use the SICL `i?peek` and `i?poke` functions to communicate with the VME devices. With these functions, you needed to know which register you want to communicate with and the register's offset. See the instrument's user's manual for a description on the registers and register locations.

The following is an example of using `iwpeek`:

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

See Chapter 10 for a complete description of the `i?peek` and `i?poke` functions.

Unmapping Memory Space

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

VME Interrupts

There are seven VME interrupt lines that can be used. By default, VXI processing of the IACK value will be used. However, if you configure VME IRQ lines and `VME Only`, no VXI processing of the IACK value will be done. That is the IACK value will be passed to a SICL interrupt handler directly. See the *I/O Libraries Installation and Configuration Guide* for information on configuring for `VME Only`. Also see `isetintr` in Chapter 10 for information on the VME interrupts.

VME Example

When you have a VME device that requires A16 address space that is accessed via an E1482 VXI-MXI Extender Bus card, you need to make an entry in the `/etc/opt/sicl/vxi16/oride.cf` file to open an A16 address window. The following is an example entry that opens a 512 byte window in A16 address space starting at address 0x7000, with the E1482 at logical address 1:

```
1  0xC  0x6770
```

When you have a VME device that requires A24 or A32 address space, you need to make an entry in the `/etc/opt/sicl/vxi16/vmedev.cf` file to reserve the appropriate address range. The following is an example entry for a VME device in slot 6 of a VXI cardcage. The cardcage is accessed by an embedded controller or top-level MXI bus. The device requires 4096 bytes of A24 address space starting at address 0x400000 and uses IRQ line 3:

```
vmedev1   0  6  A24  0x400000  0x1000  3
```

Where `vmedev1` is the name of the device, `0` is the logical address of the device through which the VXI resource manager will access the bus, `6` is the VXI slot number, `A24` is the address space to map the VME registers, `0x400000` is the starting address, `0x1000` is the size, and `3` is the irq line.

Note If your VME device requires both A24 and A32 address space, you will need to have an entry for each address space. Each line should use a different device name (for example, vmedev1 and vmedev2).

Once you have made the appropriate entry into the `vmedev.cf` file you must re-run the Resource Manager.

The following ANSI C example program opens a VXI/MXI interface session and sets up an interrupt handler. When the `I_INTR_VME_IRQ1` interrupt occurs, the function defined in the interrupt handler will be called. The program then writes to the registers, causing the `I_INTR_VME_IRQ1` interrupt to occur. Note that you must edit this program to specify the starting address and register offset of your specific VME device. This example program also requires the VME device to be using

`I_INTR_VME_IRQ1` and the VXI controller to be the handler for the VME IRQ1.

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

#define ADDR "vxi"

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Looking at SICL Function Support with VXI/MXI

This section describes how SICL functions are implemented for VXI/MXI sessions.

Device Sessions

Message-Based Device Sessions

The following describes how some SICL functions are implemented for VXI/MXI device sessions (for message-based devices):

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

Register-Based Device Sessions

Because *register-based* devices do not support the word serial protocol, and other features of *message-based* devices, the following SICL functions are not supported with register-based device sessions:

- *Non-formatted I/O*
  - `iread`
  - `iwrite`
  - `itermchr`

- *Formatted I/O*
  - `iprintf`
  - `iscanf`
  - `ipromptf`
  - `ifread`
  - `ifwrite`
  - `iflush`
  - `isetbuf`
  - `isetubuf`
- *Device/Interface Control*
  - `iclear`
  - `ireadstb`
  - `isetstb`
  - `itrigger`
- *Service Requests*
  - `igetonsrq`
  - `ionsrq`
- *Timeouts*
  - `igettimeout`
  - `itimeout`
- *VXI Specific*
  - `ivxiws`

All other functions will work with all VXI/MXI devices (message-based, register-based, etc.)

Use the `i?peek` and `i?poke` functions to communicate with register-based devices.

Interface Sessions

The following describes how some SICL functions are implemented for VXI/MXI interface sessions:

iwrite and iread	Not supported for VXI/MXI interface sessions and return the `I_ERR_NOTSUPP` error.
iclear	Causes the VXI/MXI interface to perform a SYSREST on interface sessions. Note that this will cause all VXI/MXI devices to reset.

Using SICL Trigger Lines

VXI controller can implement a subset of the trigger lines supported by SICL. See the documentation that came with your VXI controller for specifics. These values may be passed to the `ivxitrig` or `isetintr` function:

Trigger Lines

SICL
I_TRIG_TTL0
I_TRIG_TTL1
I_TRIG_TTL2
I_TRIG_TTL3
I_TRIG_TTL4
I_TRIG_TTL5
I_TRIG_TTL6
I_TRIG_TTL7
I_TRIG_ECL0
I_TRIG_ECL1
I_TRIG_ECL2
I_TRIG_ECL3
I_TRIG_EXT0
I_TRIG_EXT1
I_TRIG_EXT2
I_TRIG_EXT3
I_TRIG_CLK0
I_TRIG_CLK1
I_TRIG_CLK2
I_TRIG_CLK10
I_TRIG_CLK100

* The I_TRIG_CLK0 is the internal 16 MHz clock. This trigger line can *ONLY* be routed out.

The `itrigger` function, when used on a VXI/MXI interface session, generates the same results as the ixtrig functions with the `I_TRIG_STD` value passed to it.

The `I_TRIG_STD` value, when passed to the `ixtrig` function causes one or more VXI trigger lines to fire. The trigger lines represented by `I_TRIG_STD` are determined by the `ivxitrigroute` function. The `I_TRIG_STD` value has no default value. Therefore, if it is not defined before it is used, no action will be taken.

The following is an example that illustrates how to use some of the SICL VXI trigger functions:

```
/* trigger.c
   An example program illustrating various trigger
   operations with SICL*/

#include <sicl.h>
#include <unistd.h>

main()
{
   INST id;

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

   /*Open a vxi interface session*/
   id = iopen("vxi");

   /*Assert (drive low) TTLTRG2, TTLTRG4, and TTLTRG6 for 1 sec*/
   ivxitrigon(id, I_TRIG_TTL2  I_TRIG_TTL4  I_TRIG_TTL6);
   sleep(1);

   /*De-Assert (drive high) all previously asserted trigger lines*/
   ivxitrigoff(id, I_TRIG_ALL);

   /*Route External Trigger In SMB Connector (EXT0) to TTLTRG0*/
   ivxitrigroute(id, I_TRIG_EXT0, I_TRIG_TTL0);

   /*Route internal clock to External Trigger Out SMB
     Connector (EXT1)*/
   ivxitrigroute(id, I_TRIG_CLK0, I_TRIG_EXT1);

   /*Turn off previous routing*/
   ivxitrigroute(id, I_TRIG_EXT0, 0);
   ivxitrigroute(id, I_TRIG_CLK0, 0);

   /*Set up I_TRIG_STD routing to TTLTRG1 and TTLTRG3*/
   ivxitrigroute(id, I_TRIG_STD, I_TRIG_TTL1  I_TRIG_TTL3);

   /*Fire the STD triggers*/
   ixtrig(id, I_TRIG_STD);

   /*Close the vxi interface session*/
   iclose(id);
}
```

Using i?blockcopy for DMA Transfers

VXI controllers can have the capability for block copy DMA transfers. This can be done using the SICL i?blockcopy functions. Use the following process to access DMA transfers:

1. Use the SICL `imap` function to map the desired VXIbus address.
2. Use the SICL `itimeout` function to set up a timeout value.
3. Use the SICL `i?blockcopy` function to initiate the DMA transfer. Note that the `swap` parameter is ignored.

The following example illustrates using ibblockcopy for a DMA transfer:

Note SICL does not support overlapped DMA transfers, which means the `i?blockcopy` functions will not return until the end of the DMA transfer

```
/* blockcopy.c
   This example demonstrates how to use i?blockcopy to
   move data.  The SICL blockcopy routines will attempt to
   use DMA, if one of the locations is A24 or A32 address
   space.  If neither location is in A24 or A32 space the
   data will be move in the normal fashion.

   Usage:
     blockcopy -a <symbolic_name>
   Return Value:
     none */




#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <sicl.h>

extern char *optarg;
static void error_usage(const char *);

main(int argc, char *argv[]) {
   long o;
   INST id;
   static char *a24_buf;
   static char *shr_buf;
   unsigned long bufsize = 1024 * 2;
   char *addr = NULL;

   while ((o = getopt(argc, argv, "a:b:i:n:")) != EOF)
      switch (o) {
      case 'a':
         addr = optarg;
         break;
      default:
         error_usage(argv[0]);
         break;
      }

   if (addr == NULL)
      error_usage(argv[0]);

   ionerror (I_ERROR_NO_EXIT);
   id = iopen (addr);

   shr_buf = imap (id, I_MAP_SHARED, 0, 0, 0);
   a24_buf = imap (id, I_MAP_A24, 0x20, 0x8, 0);

   printf("Shared memory to A24 (D16).\n\n");
   iwblockcopy (id,
                unsigned short *)shr_buf,
                unsigned short *)a24_buf,
                bufsize,
                0
                );
   printf("A24 to Shared memory (D16).\n\n");
   iwblockcopy (id,
                (unsigned short *)a24_buf,
```

```
                (unsigned short *)shr_buf,
                1,
                0
                );

  printf("Shared memory to A24 (D32).\n\n");
  ilblockcopy (id,
              (unsigned long *)shr_buf,
              (unsigned long *)a24_buf,
              bufsize,
              0
              );

  printf("A24 to Shared memory (D32).\n\n");
  ilblockcopy (id,
              (unsigned long *)a24_buf,
              (unsigned long *)shr_buf,
              bufsize,
              0
              );
}

static void error_usage(const char *progname)
{
  printf("Usage Error: %s <options>\n", progname);
  printf("\t-a <addr>:\tSICL address\n");
  exit(1);
}
```

Using VXI Specific Interrupts

See the `isetintr` function in Chapter 10 for a list of VXI/MXI specific interrupts.

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

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

Processing VME Interrupts Example

```
/* vmeintr.c
   This example uses SICL to cause a VME interrupt from an
   HP E1361 register-based relay card at logical address 136.*/
#include <sicl.h>

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

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

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

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

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

Summary of VXI/MXI Specific Functions

Note Using these VXI interface specific functions means that the program can not be used on other interfaces and, therefore, becomes less portable.

These functions will work over a LAN-gatewayed session if the server supports the operation.

SICL VXI/MXI Functions

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

7

Using SICL with RS-232

Using SICL with RS-232

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

This chapter explains how to use SICL to communicate over RS-232. In order to communicate over RS-232, you must have loaded the RS232 fileset during the I/O Libraries installation. See the *I/O Libraries Installation and Configuration Guide* for information. Also note that the RS-232 related SICL functions have the string `SERIAL` embedded in the functions' names.

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 `/opt/sicl/share/examples` directory.

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 GPIB and VXI, 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 GPIB.

Using SICL to communicate with an instrument on RS-232 is similar to using SICL over GPIB. 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.

Communicating with RS-232 Devices

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 either the interface `symbolic name` or `logical unit` followed by a device logical address of `488` in the *addr* parameter of the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values. 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 titled "SICL Function Support with RS-232 Device Sessions."

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

`COM1,488`	A RS-232 device connected to COM1.
`COM2,488`	A RS-232 device connected to COM2.

Note The previous examples use the default `symbolic name` specified during the system configuration. If you want to change the name listed above, you must also change the `symbolic name` or `logical unit` specified during the configuration. The name used in your SICL program must match the `logical unit` or `symbolic name` specified in the system configuration. Other possible interface names are `serial`, `SERIAL`, etc.

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.

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

SICL Function Support with RS-232 Device Sessions

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

`iprintf, iscan, 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 "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 supports 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 supports 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 "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 Chapter 10 for information on the device session interrupts for RS-232.

RS-232 Device Session Example

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

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

main()
{
   INST dvm;
   double res;

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

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

   /* Reset the multimeter */
   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);
}
```

Communicating with RS-232 Interfaces

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. When communicating with an RS-232 interface session, you specify the interface name.

Addressing RS-232 Interfaces

To create an interface session on RS-232, specify either the interface `symbolic name` or `logical unit` and a particular device's address in the *addr* parameter of the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

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

`COM1`	An interface symbolic name.
`COM2`	An interface symbolic name.
`9`	An interface logical unit.

Note The previous examples use the default `symbolic name` specified during the system configuration. If you want to change the name listed above, you must also change the `symbolic name` or `logical unit` specified during the configuration. The name used in your SICL program must match the `logical unit` or `symbolic name` specified in the system configuration. Other possible interface names are `serial`, `SERIAL`, etc.

The following example opens an interface session with the RS-232 interface.

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

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 control 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`	Same as `ixtrig` (`I_TRIG_STD`). 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, 1)`

`ionsrq`

Installs a service request handler for this session. The concept of service request (SRQ) originates from GPIB. On an GPIB interface, a device can request service from the controller by asserting a line on the interface bus. RS-232 does not have a specific line assigned as a service request line. 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.

`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. • `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_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 the hardware receive buffer, not how much data is in the buffer used by the formatted input routines such as `iscanf`.
iserialmclctrl	Controls the modem control lines RTS and DTR. If one of these lines is being used for flow control, you cannot set that line with this function.
iserialmclstat	Determines the current state of the modem control lines. If one of these lines is being used for flow control, this function may not give the correct state of that line.

RS-232 Interface Session Interrupts

There are specific interface session interrupts that can be used. See `isetintr` in Chapter 10 for information on the interface session interrupts for RS-232.

RS-232 Interface Session Example

```
/* serialintf.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;

   /* 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);

  return 0;
}
```

Summary of RS-232 Specific Functions

Note Using these RS-232 interface specific functions means that the program can not be used on other interfaces and, therefore, becomes less portable.

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_EVEN I_SERIAL_PAR_ODD
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 SICL with LAN

Using 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. In order to communicate over the LAN, you must have loaded the LAN fileset during installation for a host system acting as a LAN client, and you must have loaded the LANSVR fileset during installation for a host system acting as a LAN server. See the *I/O Libraries Installation and Configuration Guide* for information. The example programs shown in this chapter are also provided in the `/opt/sicl/share/examples` directory.

This chapter contains the following sections:

- Overview of SICL LAN
- Considering LAN Configuration and Performance
- Communicating with Devices over LAN
- Using Timeouts with LAN
- Using Signal Handling with LAN
- Summary of LAN Specific Functions

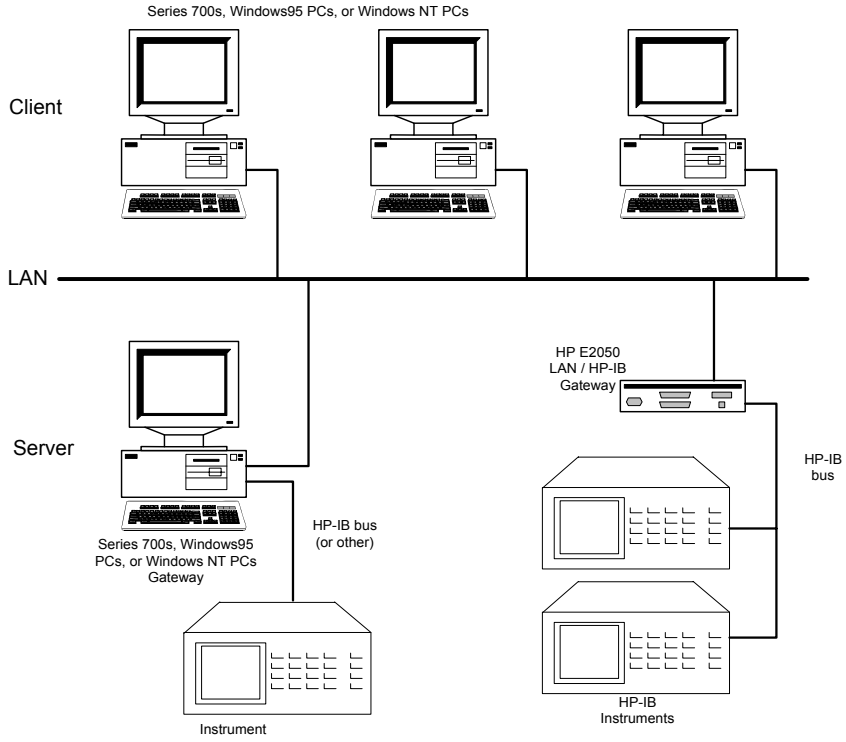
Overview of SICL LAN

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 (such as a Series 700 HP-UX Workstation) makes SICL requests over the network to a LAN server (such as a Series 700 HP-UX workstation, a PC, or an HP/Agilent E2050 LAN/GPIB Gateway). The LAN server is connected to the instrumentation or devices that must be controlled. Once the LAN server has completed the requested operation on the instrument or device, the LAN server sends a reply to the LAN client. This reply contains any requested data and status information which indicates whether the operation was successful.

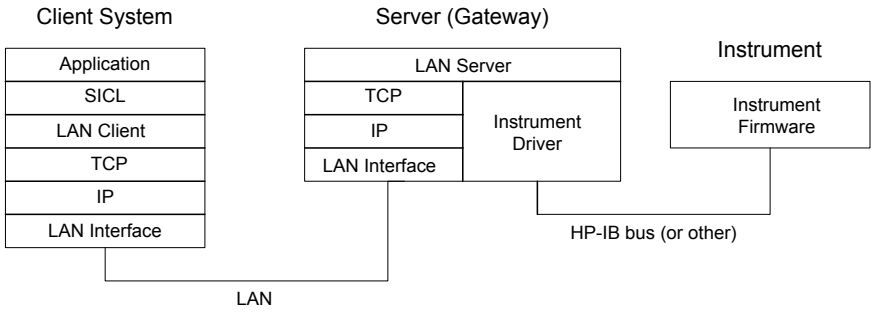


Using the LAN Client and LAN Server (Gateway)

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 (`SICL-LAN` fileset) and the LAN software (TCP/IP) needed to access the server (gateway). The gateway contains the LAN server software(`SICL-LANSVR` fileset), 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 `iosetup` utility) to use SICL over LAN. The two protocols are as follows:

- **SICL LAN Protocol** is a networking protocol developed by HP/Agilent which is compatible with all existing SICL LAN products. This LAN networking protocol is the default choice in the `iosetup` utility when you are configuring LAN for SICL.

- **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 Unix currently supports SICL operations over the LAN to GPIB/GPIB and VXI interfaces. Also, some SICL operations are not supported when using the TCP/IP Instrument Protocol. See the section titled "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 GPIB.

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 `iosetup` utility. (See the *I/O Libraries Installation and Configuration Guide* for information on running `iosetup`.) 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.

SICL LAN Server

SICL includes the necessary software to allow a Series 700 workstation or Linux PC to act as a LAN-to-instrument_interface gateway. The fileset `SICL-LANSVR`, provides a daemon, `siclland`, which will accept I/O requests from a SICL LAN client and perform the I/O operations on a local interface.

To use this capability, the Series 700 or Linux PC must have a local interface configured for I/O. The supported interfaces for this release are GPIB, VXI/MXI, and RS-232 for the SICL LAN protocol and GPIB/GPIB and VXI interfaces for the TCP/IP Instrument Protocol. See the "SICL Function Support with LAN-gatewayed Sessions" section later in this chapter for information on which functions are not supported over LAN.

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 TAMS representative for a current list of other supported SICL LAN servers.

Considering LAN Configuration and Performance

As with other client/server applications on a LAN, when deploying an application which uses SICL LAN, consideration must be given to the performance and configuration of the network the client and server will be attached to. 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 it is simply a general design consideration when developing 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 Devices over LAN

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 "SICL Function Support with LAN-gatewayed Sessions" section later in this chapter). This means most operations you might request of an interface, such as GPIB, connected directly to your controller, you can 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 SICL LAN server on the network.

Addressing Devices or Interfaces with LAN-gatewayed Sessions

To create a LAN-gatewayed session, specify the LAN interface `symbolic name` or `logical unit`, 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 `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

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]:GPIB,7`	A device address corresponding to the device at primary address 7 on the GPIB interface attached to the machine named `instserv` in the `hp.com` domain (Fully qualified domain names may be used).
`lan[128.10.0.3]: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 with IP address `128.10.0.3`.
`lan[intserv]:GPIB`	An interface address corresponding to the GPIB 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` (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

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 functions are not supported over LAN:

- `i?blockcopy`
- `imap`
- `imapinfo`
- `i?peek`
- `i?poke`
- `i?popfifo`
- `i?pushfifo`
- `iunmap`

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

- All VXI specific functions
- All RS-232/serial specific functions
- `igetlu`
- `ionintr`
- `isetintr`
- `igetintfsess`
- `igetonintr`
- `igpibgett1delay`
- `igpibllo`
- `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.

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 specific functions
- All VXI specific functions
- All 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 program opens an GPIB device session via a LAN-to-GPIB gateway. Note that this example is the same as the first example in the "Using SICL with GPIB" chapter, only the addresses passed to the `iopen` calls are modified. The example addresses assume the machine with hostname `instserv` is acting as a LAN-to-GPIB gateway.

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

main()
{
   INST dvm;
   INST sw;

   double res;
   int i;

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

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

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

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

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

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

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

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

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 either the interface `symbolic name` or `logical unit` and a particular device's address in the *addr* parameter of the `iopen` function. The interface `symbolic name` and `logical unit` are defined during the system configuration. See the *I/O Libraries Installation and Configuration Guide* for information on these values.

The following are examples of LAN interface addresses:

`lan`	A LAN interface address
`30`	A LAN interface address (30 is the default lu for LAN)

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 GPIB specific functions
- All VXI 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 SICL LAN gateways, 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 Timeouts with LAN

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 configuration values specified during the system configuration (see the *I/O Libraries Installation and Configuration Guide* for information on setting this value). Once `ilantimeout` is called by the application, the automatic LAN timeout adjustment described in the next sub-section is turned off. See Chapter 10 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 Chapter 10.

Default LAN Timeout Values

The LAN Client interface configuration 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	imeout 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.

See the *I/O Libraries Installation and Configuration Guide* for information on setting these values.

Note Once `ilantimeout` is called, the software no longer sends the server timeout 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 during the SICL LAN interface configuration 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 SICL LAN applications which you want to reconfigure.

2. As `root`, run the `iosetup` utility and edit the LAN interface. Change the Server Timeout or Client Timeout Delta parameter. (See the *I/O Libraries Installation and Configuration Guide* for information on changing these values.

3. Restart the SICL LAN applications.

When only reconfiguring the LAN interface, note that you do not need to rebuild the kernel for changes to take effect.

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 in the LAN interface configuration. 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.

Note that another way to ensure that the server does not wait forever is via the `-t` *timeout* parameter to the `siclland` daemon. By default, `siclland` will use a 2 minute timeout if a timeout value of infinity is received from the client.

Application Terminations and Timeouts

If an application is killed either via **Ctrl-C** or the `kill` command 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 supplied LAN server , 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) or the LAN server (via the Server Timeout value) may be configured to use a shorter timeout value.

If long timeouts must be used, the server may be reset. A server may be reset by logging into the server host and killing the running `siclland` daemon(s). Note that the latter procedure will affect all clients connected to the server. See the LAN section in Chapter 9, "Troubleshooting Your 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.

Using Signal Handling with LAN

SIGIO Signals

SICL uses SIGIO for SRQs and interrupts on LAN interfaces. The SICL LAN client installs a signal handler to catch SIGIO signals. To enable sharing of SIGIO signals with other portions of an application, the SICL LAN SIGIO signal handler remembers the address of any previously installed SIGIO handler, and calls this handler after processing a SIGIO signal itself. If your application installs a SIGIO handler, it should also remember the address of a previously installed handler and call it before completing.

The signal number used with LAN (SIGIO) can *not* be changed. Note that `isetsig()` has no effect on LAN.

However, if you must share SIGIO or any signal set with isetsig() between SICL and another portion of your application, your application must adhere to the following guidelines. These guidelines allow for multiple signal handlers to be called when a signal is received.

- Store the address of the previously installed signal handler when installing your signal handler. Call this stored handler address when a signal is received.
- Note that both SIG_DFL and SIG_IGN may be returned as "previous" handlers, and an application may need to deal with these as necessary.
- Handle spurious signals (that is, signals intended for the previous handler or other portions of your application).
- Install a signal handler once per process, and *never* remove the handler.
- Don’t block signals by default. (However, blocking/unblocking around short, critical operations is okay.)
- Use `sigaction()` to install signal handlers. Other signal handling mechanisms supported by Unix are not compatible with SICL, which uses `sigaction()`.

SIGPIPE Signals

The SICL LAN client also installs a signal handler for SIGPIPE. This ensures that a broken network connection will not cause the SICL application to terminate or exit unexpectedly.

The SICL LAN client does no special processing when it receives a SIGPIPE signal, but will pass the signal along to a previously installed SIGPIPE handler unless configured not to during the SICL configuration. If an application installs a SIGPIPE handler, it should chain handlers in the same manner as described for SIGIO.

Summary of LAN Specific Functions

Note Using these LAN interface specific functions means that the program can not be used on other interfaces and, therefore, becomes less portable.

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 SICL Program

Troubleshooting Your SICL Program

This chapter provides a guide to troubleshooting errors that may occur when using SICL.

This chapter contains the following sections:

- Installing an Error Handler
- Looking at Error Codes and Messages
- Troubleshooting SICL
- Troubleshooting SICL over LAN
- Troubleshooting SICL over RS-232
- Troubleshooting SICL over GPIO
- Where to Find Additional Information

Installing an Error Handler

One of the simplest ways to detect SICL run-time errors is to install an error handler. SICL allows you to install an error handler for all SICL functions within an application. When a SICL function call results in an error, the error routine specified in the error handler is called. You can use one of the error routines provided by SICL, or you can write your own error routine.

Use the SICL `ionerror` function to install an error handler:

```
ionerror (proc);
```

Where *proc* is the error routine to be called when a SICL function call results in an error. The following are error routines provided by SICL:

`I_ERROR_EXIT`	This value installs a special error handler which will print a diagnostic message and then terminate the **process**.
`I_ERROR_NO_EXIT`	This value installs a special error handler which will print a diagnostic message and then allow the process to continue execution.

See "Using Error Handlers" in Chapter 3 of this manual for more information on installing a SICL error handler and writing your own error routine. You can also see Chapter 10 for details about the `ionerror` function call.

Looking at Error Codes and Messages

When you install a default SICL error routine such as `I_ERROR_EXIT` or `I_ERROR_NOEXIT` with an `ionerror` call, the SICL error message is printed.

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.

The following table contains an alphabetical summary of SICL error messages:

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 in the `hwconfig`.cf file.
`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, etc. imply invalid data.
`I_ERR_INTERNAL`	`Internal error occurred`	SICL internal error.
`I_ERR_INTERRUPT`	`Process interrupt occurred`	A process interrupt has occurred in your application.
`I_ERR_INVLADDR`	`Invalid address`	The address specified in iopen is not a valid address (e.g. "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 intrinsic).
`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.

Error Codes and Messages (Continued)

Error Code	Error String	Description
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 that is newer than the drivers. Need to update drivers.

Troubleshooting SICL

When using SICL you typically have to go through a compile/link process and then run the program. You can get errors in either of these steps. This section is divided into two subsections:

- Compile and Link Errors
- Run-time Errors

Compile and Link Errors

Compile Errors - Unexpected symbol

You get a list of errors where the compiler doesn't recognize SICL symbols. For example:

```
cc: "example.c", line 12 : error 1000: Unexpected symbol:  "id".
cc: "example.c", line 12: error 1573: Type of "id" is undefined.
cc: "example.c", line 16: error 1588: "I_ERROR_EXIT" undefined.
cc: "example.c":, line 19: error 1549: Modifiable lvalue required
         for assignment operator.
```

Possible Solution. This error indicates that some of the SICL declarations are undefined during the compile process. Check to make sure you added the `sicl.h` header file. Use the `#include` command at the beginning of your program followed by the `sicl.h` header file. See "Compiling and Linking an SICL Program" in Chapter 2 for more information.

Link Errors - Unsatisfied symbols

The linker doesn't recognize the SICL function calls. For example:

```
/bin/ld : Unsatisfied symbols:
   I_ERROR_EXIT (code)
   iclose (code)
   ipromptf (code)
   ionerror (code)
   iopen (code)
      .
      .
      .
```

Possible Solution. This error indicates that the SICL functions are not being found during the link process. Most likely, you have left out the SICL library during the link process. Link in the SICL library with the `-lsicl` option during the compile/link process.

Compile/Link Error - Undefined id

SICL assignments are undefined. For example:

```
"example.c", line 10 : error 1588 "id" undefined
"example.c", line 10 : error 1549 Modifiable lvalue
                       required for assignment operator
```

Possible Solution. This error indicates that one of your assignments is undefined. Check to make sure you declared your session as a SICL type INST at the beginning of your program. Include an `INST id` at the beginning of your program.

Run-time Errors

Program Hangs Your program hangs while either sending or receiving data.

Possible Solution. If your SICL program hangs the first thing you should try is to add the SICL `itimeout` function. You must specify with what device or interface to time out. However, once the timeout time is reached, the call will return with the `I_ERR_TIMEOUT` error.

iopen fails - Timeout occurred `iopen` fails with a timeout error. For example:

```
ERROR hpib,22 Timeout occurred
```

Possible Solution. This error indicates that the device or interface you are trying to communicate with is not responding. Or, insufficient time was allowed for the operation, in which case a longer timeout is needed. You may be trying to communicate with a device that is not available on the bus. Check the device address.

iopen fails - Invalid Address `iopen` fails with an invalid address. For example:

```
ERROR hpib2,16 Invalid address
```

Possible Solution. This error indicates that the address specified is not valid. Several things can cause this. First of all you may be attempting to communicate with a non-existent interface. First, check that the interface name in the SICL configuration is correct. Second, you may have an invalid address. Check the address limitations. See the addressing section in the interface specific chapter.

Invalid INST Invalid `INST` when trying to communicate with a session. For example:

```
ERROR: : Invalid INST
```

Possible Solution. This error indicates that a session for the listed `INST` is not valid. Make sure you opened a communications session using the `iopen` function.

Troubleshooting SICL over LAN (Client and Server)

Before SICL 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 SICL LAN software:

- If your application is unable to open a session to the SICL LAN server, 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 might look something like the following:

```
>ping instserv.hp.com
PING instserv.hp.com: 64 byte packets
64 bytes from 128.10.0.3: icmp_seq=0. time=3. ms
64 bytes from 128.10.0.3: icmp_seq=1. time=3. ms
64 bytes from 128.10.0.3: icmp_seq=2. time=2. ms
   .
   .
```

Where each line after the `PING` line is an example of a packet successfully reaching the server. If after several seconds `ping` does not print any lines, use CTRL-C to kill `ping`. `ping` will report on what it found:

```
 ----instserv.hp.com PING Statistics----
4 packets transmitted, 0 packets received, 100% packet loss
```

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 LAN application. See the `ping(1M)` man page for more information.

- Another tool which can be used to determine where a problem might reside is `rpcinfo`. (Note that `rpcinfo` resides under the `/usr/bin` directory on HP-UX 11i, or `/usr/sbin` on Linux.) 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 the SICL LAN Protocol and `395183` which is the TCP/IP Instrument Protocol. The port number will vary. This is the `siclland` daemon line (you may or may not see the word siclland at the end of this line). If the line for program `395180` or `395183` is not present, then 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.

  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.

SICL LAN Client Problems and Possible Solutions

iopen fails - syntax errorr

`iopen` fails with error `I_ERR_SYNTAX`.

Possible Solution. 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

`iopen` fails with the error `I_ERR_BADADDR`, and the error text is `core connect failed: RPC_PROG_NOT_REGISTERED`.

Possible Solution. This indicates that the SICL LAN server 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

`iopen` fails with the error `I_ERR_SYMNAME`, and the error text is bad hostname, gethostbyname() failed.

Possible Solution. 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 utility `nslookup` can be used to determine if the hostname is known to your system. See the `nslookup(1)` man page for more information on this utility.

iopen fails - timeout

`iopen` fails with a timeout error.

Possible Solution. Increase the value of the Client Timeout Delta parameter during the SICL LAN interface configuration. See the "Using Timeouts with LAN" section in chapter 8 for more information.

iopen fails - other failures

`iopen` fails with some error other than those already mentioned above.

Possible Solution. 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, etc.)

I/O operation times out

An I/O operation times out even though the timeout being used is infinity.

Possible Solution. Increase the value of the Server Timeout value during the LAN interface configuration. Also ensure that the LAN client timeout is large enough if you used `ilantimeout`. See the "Using Timeouts with LAN" section in chapter 8 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.

Possible Solution. 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 using `ilantimeout`, you must determine and set the LAN timeout manually. Otherwise ensure that the Client Timeout Delta value specified during the LAN configuration is large enough. See "Using Timeouts with LAN" section in chapter 8 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.

Possible Solution. Old SICL LAN server processes from previous clients may not have terminated properly. Consult your server's troubleshooting documentation and follow its instructions for killing any old server processes.

SICL LAN Server Problems and Possible Solutions

rpcinfo' does not list siclland

`rpcinfo` fails to indicate that program 395180 (SICL LAN Protocol) or 395183 (TCP/IP Instrument Protocol) is available on the server.

Possible Solution. Did you run `lanconf` (`/opt/sicl/bin` directory on HP-UX 11i or Linux) as `root`? If not, do so. If so, on HP-UX 11i ensure that `/etc/rpc` and `/etc/inetd.conf` contain the following lines.

`/etc/rpc` should contain:

```
   siclland        395180
   tcpinst          395183
```

`/etc/inetd.conf` should contain:

```
   rpc stream tcp nowait root /opt/sicl/bin/siclland 395180 1
siclland -l /var/opt/sicl/siclland_log
   rpc stream tcp nowait root /opt/sicl/bin/siclland 395183 1
siclland -l /var/opt/sicl/siclland_log
```

On Linux, `/etc/init.d/siclland` should exist and

```
   /sbin/chkconfig --list siclland
```

Should show the service as being turned on for run levels 3, 4, and 5.

(Note that parameters to `siclland`, such as `-l` *logfile*, may vary depending on how you would like the server configured.)

If these entries are present, ensure that `inetd` is reconfigured to recognize the new entries by running the following as `root`:

On HP-UX 11i:

```
   /usr/sbin/inetd -c
```

On Linux:

```
   /sbin/service siclland stop ; /sbin/service siclland start
```

iopen fails

`iopen` fails when you run your application, but `rpcinfo` indicates that the LAN server is ready and waiting.

Possible Solution. Ensure that the requested interface has been configured on the server. This is done while running the I/O Libraries configuration utility. See the *I/O Libraries Installation and Configuration Guide* for more information on this configuration.

LAN server appears "hung"

The SICL LAN Server appears hung (possibly due to a long timeout being set by a client on an operation which will never succeed).

Possible Solution. Login to the LAN server (via `telnet` or `rlogin`) and kill the hung `siclland` server process. You can determine what `siclland` server processes are running by typing the following:

```
ps -ef | grep siclland
```

You'll see something like the following:

```
root  2492  2480 11 15:33:27 ?     0:00 siclland -l /var/opt/sicl/siclland_log
```

Where `2492` is the PID of the running server. You will see one server process for each client connected to this host. If more than one server is running, you have two options for killing the hung server:

- If informational logging has been enabled using the `-s` option to `siclland`, then the server process matching a client process can be determined by log entries, which by default is placed in the file `/var/opt/sicl/siclland_log`. See `siclland(1m)` for details.
- If no logging has been enabled, then the server as a whole will need to be reset by killing all `siclland` processes. Note that this will break the connections to all clients, even those which are still operational.

  Use the following to kill a LAN server process. This must be done as root:

  `kill` *PID_number*

rpcinfo fails - can't contact portmapper

`rpcinfo` returns the message `rpcinfo`: can't contact portmapper: `RPC_SYSTEM_ERROR - Connection refused`.

Possible Solution. Ensure that the portmapper is running on the server. See `portmap(1m)` for details on starting the portmapper.

Note that if you must restart the portmapper, you must then reconfigure `inetd` by running the following as `root`:

On HP-UX 11i:

```
/usr/sbin/inetd -c
```

On Linux:

```
kill -HUP `cat /var/run/inetd.pid`
```

rpcinfo fails - programs 395180 or 395183 are not available

`rpcinfo -t` *server_hostname* `395180 1` OR `rpcinfo -t` *server_hostname* `395183 1` returns the following message:

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

Possible Solution. Ensure that `inetd` is running on the server. See `inetd(1m)` for details on starting `inetd`.

Troubleshooting SICL over RS-232

Unlike GPIB, 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. Use of a RS-232 protocol analyzer may be of assistance.

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 *I/O Libraries Installation and Configuration Guide*, as well as to the *RS-232 Cables* insert included in your 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.

RS-232 Port Allocation and HP-UX termio Functions

Note that an RS-232 port which is configured for use by SICL is not available for use by Unix `termio` functions, and vice-versa.

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

Troubleshooting SICL over GPIO

Because the GPIO interface has such flexibility, most initial problems come from cabling and configuration. There are many fields that be specified during the I/O Libraries configuration. 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 configuration values 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/Agilent E2074 cabling and handshake behavior. In either case, it is important to read the *HP/Agilent E2074 GPIO Interface Installation Guide*.

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/Agilent E2074 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 the `Symbolic Name` specified during the configuration. 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, first check the values in your GPIO interface configuration values and ensure that the configuration was processed successfully. As `root`, execute the command:

On HP-UX 11i:

```
/sbin/dmesg
```

On Linux:

```
/bin/dmesg
```

If there were no errors, the I/O configuration section of `dmesg` will contain a line such as:

```
SICL: HP E2074 GPIO: Initialized ...
```

If there was a problem, you will see a short diagnostic message containing the words `GPIO config`. This diagnostic message will help you identify the field in the SICL configuration which contained the error.

Operation Not Supported

The HP/Agilent E2074 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 by running the configuration utility. See the *I/O Libraries Installation and Configuration Guide* for information on running this 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.

Generic I/O Error

This error results if you have specified `I_HINT_USEDMA` and also specified other conditions that are inconsistent with DMA. For example, the DMA controller cannot perform pattern matching. So setting `itermchr` or `I_GPIO_READ_EOI` prevents the use of DMA.

The easiest way to avoid this error is to avoid the use of `ihint`. The system always picks an appropriate mode for any transaction, if left to its own devices. If you believe that `I_HINT_USEDMA` is needed in your program, be careful to avoid any other requirements or conditions that prevent the use of DMA.

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.

Where to Find Additional Information

For Compile/Link Errors see the following:

- Chapter 2, "Getting Started with SICL," for SICL compile/link instructions.
- Chapter 3, "Using SICL," for a description of how to use SICL.
- HP *C Programmers Guide* to review usage of pointers and pointer types.

For Run-time Errors see the following:

- Chapter 3, "Using SICL," for a description of SICL features.
- The interface specific chapter for a description of valid addressing.
- *I/O Libraries Installation and Configuration Guide* for a description of the I/O Libraries configuration process.
- HP *C Programmers Guide* to review usage of pointers and pointer types.

For LAN problems see the following:

- Chapter 8, "Using SICL with LAN," for a description of LAN addressing and timeouts.
- Your network administrator.
- The *Installing and Administering LAN/9000 Software* manual for HP-UX.

10

SICL Language Reference

SICL Language Reference

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

- C syntax.
- Complete description.
- Return value(s).
- Related SICL functions that you may want to see, also.

Session Identifiers

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

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

```
INST id;
```

Device, Interface, and Commander Sessions

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

Functions Affected by Locks

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

Functions Affected by Timeouts

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

Per-Process Functions

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

IABORT

Supported Sessions . . . . . device, interface, commander

C Syntax

```
#include <sicl.h>

int iabort (id);
inst id;
```

Note This function is *only* supported with C/C++.

Also, this function has no effect over LAN for any of the LAN servers, such as the HP/Agilent E2050 LAN/GPIB Gateway.

Description The `iabort` function will abort any SICL calls currently executing with the current session *id*, regardless of what **thread** it is executing on. However, since session *ids* are only valid within a single process, only SICL calls in progress in the current process will be affected. The SICL call being aborted will return the error code `I_ERR_ABORTED`, implying that it was aborted by another **thread**. If no **thread** has any SICL calls pending on the given session *id*, this function will perform no action.

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

IBLOCKCOPY

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

C Syntax

```
#include <sicl.h>

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

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

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

Note Not supported over LAN.

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

The *id* parameter, although specified, is normally ignored except to determine an interface-specific transfer mechanism such as DMA. To prevent using an interface-specific mechanism, pass a zero (`0`) for this parameter. The *src* argument is the starting memory address for the source data. The *dest* argument is the starting memory address for the destination

data. The *cnt* argument is the number of transfers (bytes, words, or long words) to perform. The *swap* argument is the byte swapping flag. If swap is zero, no swapping occurs. If *swap* is non-zero the function swaps bytes (if necessary) to change byte ordering from the internal format of the controller to/from the VXI (big-endian) byte ordering.

Note If a bus error occurs, unexpected results may occur.

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

See Also "IPEEK", "IPOKE", "IPOPFIFO", "IPUSHFIFO"

IBLOCKMOVEX

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

C Syntax

```
#include <sicl.h>

int iblockmovex (id, src_handle, src_offset, src_width,
                 src_increment, dest_handle, dest_offset,
                 dest_width, dest_increment, cnt, swap);
   INST id;
   unsigned long src_handle;
   unsigned long src_offset;
   int src_width;
   int src_increment;
   unsigned long dest_handle;
   unsigned long dest_offset;
   int dest_width;
   int dest_increment;
   unsigned long cnt;
   int swap;
```

Note Not supported over LAN.

Note If either the *src_handle* or the *dest_handle* is NULL, then the handle is assumed to be for local memory. In this case, the corresponding offset is a valid memory address.

Description `iblockmovex` moves data (8-bit byte, 16-bit word, and 32-bit long word). from memory on one device to memory on another device. This function allows local-to-local memory copies (both *src_handle* and *dest_handle* are zero), VXI-to-VXI memory transfers (both *src_handle* and *dest_handle* are valid handles), local-to-VXI memory transfers (*src_handle* is zero, *dest_handle* is valid handle), or VXI-to-local memory transfers (*src_handle* is valid handle, *dest_handle* is zero).

The *id* parameter is the value returned from `iopen`. If the *id* parameter is zero (0) then all handles must be zero and all offsets must be either local memory or directly dereferencable VXI pointers.

The *src_handle* argument is the starting memory address for the source data. The *dest_handle* argument is the starting memory address for the destination data. These handles must either be valid handles returned from the `imapx` function (indicating valid VXI memory), or zero (0) indicating local memory. Both *src_width* and *dest_width* must be the same value; they specify the width (in number of bits) of the data. Specify them as 8, 16, or 32. Offset values (*src_offset* and *dest_offset*) are generally used in memory transfers to specify memory locations. The increment parameters specify whether or not to increment memory addresses. The *cnt* argument is the number of transfers (bytes, words, or long words) to perform. The *swap* argument is the byte swapping flag. If swap is zero, no swapping occurs. If *swap* is non-zero the function swaps bytes (if necessary) to change byte ordering from the internal format of the controller to/from the VXI (big-endian) byte ordering.

Note If a bus error occurs, unexpected results may occur.

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

See Also "IPEEKX8, IPEEKX16, IPEEKX32", "IPOKEX8, IPOKEX16, IPOKEX32", "IPOPFIFO", "IPUSHFIFO", "IDEREFPTR"

ICAUSEERR

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

C Syntax

```
#include <sicl.h>

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

VDescription Occasionally it is necessary for an application to simulate a SICL error. The `icauseerr` function performs that function. This function causes SICL to act as if the error specified by *errcode* (see Chapter 9, Troubleshooting Your SICL Program, for a list of errors) has occurred on the session specified by *id*. If *flag* is 1, the error handler associated with this process is called (if present); otherwise it is not.

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

See Also "IONERROR", "IGETONERROR", "IGETERRNO", "IGETERRSTR"

ICLEAR

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

C Syntax

```
#include <sicl.h>

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

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

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

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

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

See Also "IFLUSH", and the interface-specific chapter in this manual for details of implementation.

ICLOSE

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

C Syntax

```
#include <sicl.h>

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

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

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

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

See Also "IOPEN"

IDEREFPTR

Supported Sessions: . . . . . device, interface, commander

C Syntax

```
#include <sicl.h>

int idereptr (id, handle, *value);
   INST id;
   unsigned long handle;
   unsigned char *value;
```

Description This function tests the handle returned by `imapx`. The *id* is the valid SICL session id returned from the `iopen` function, *handle* is the valid SICL map handle obtained from the `imapx` function. This function sets **value* to zero (0) if `imap` or `imapx` returns a map handle that cannot be used as a memory pointer; you must use `ipeekx`*8*, `ipeekx16`, `ipeekx32`, `ipokex8`, `ipokex16`, `ipokex32`, or `iblockmovex` functions. Alternately, the function returns a non-zero value if `imapx` returns a valid memory pointer that can be directly dereferenced.

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

See Also "IMAPX", "IUNMAPX", "IPEEKX8, IPEEKX16, IPEEKX32", "IPOKEX8, IPOKEX16, IPOKEX32", "IBLOCKMOVEX"

IFLUSH

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

C Syntax

```
#include <sicl.h>

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

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

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

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

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

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

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

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

IFREAD

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

C Syntax

```
#include <sicl.h>

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

Description

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

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

Values for *reason* include:

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

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

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

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

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

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

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

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

IFWRITE

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

C Syntax

```
#include <sicl.h>

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

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

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

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

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

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

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

IGETADDR

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IOPEN"

IGETDATA

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

C Syntax

```
#include <sicl.h>

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

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

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

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

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

See Also "ISETDATA"

IGETDEVADDR

Supported sessions: . . . . . device

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IOPEN"

IGETERRNO

C Syntax

```
#include <sicl.h>

int igeterrno ();
geterrno ()
```

Description All functions (except a few listed below) return a zero if no error occurred (`I_ERR_NOERROR`), or a non-zero error code if an error occurs (see Chapter 9, Troubleshooting Your SICL Program, for a list of errors). This value can be used directly. The `igeterrno` function will return the last error that occurred in one of the library functions.

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

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

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

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

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

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

See Also "IONERROR", "IGETONERROR", "IGETERRSTR", "ICAUSEERR"

IGETERRSTR

C Syntax

```
#include <sicl.h>

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

Description SICL has a set of defined error messages that correspond to error codes (see Chapter 9, Troubleshooting Your SICL Program, for a list of errors) that can be generated in SICL functions. To get these error messages from error codes, use the `igeterrstr` function.

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

See Also "IONERROR", "IGETONERROR", "IGETERRNO", "ICAUSEERR"

IGETGATEWAYTYPE

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

C Syntax

```
#include <sicl.h>

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

Description

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

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

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

Return Value

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

See Also

Chapter 8 - Using SICL with LAN

IGETINTFSESS

Supported sessions: . . . . . device, commander

C Syntax

```
#include <sicl.h>

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

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

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

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

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

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

See Also "IOPEN"

IGETINTFTYPE

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

C Syntax

```
#include <sicl.h>

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

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

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

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

See Also "IOPEN"

IGETLOCKWAIT

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "ILOCK", "IUNLOCK", "ISETLOCKWAIT"

IGETLU

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IOPEN", "IGETLUINFO"

IGETLUINFO

C Syntax

```
#include <sicl.h>

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

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

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

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

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

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

See Also "IOPEN", "IGETLU", "IGETLULIST"

IGETLULIST

C Syntax

```
#include <sicl.h>

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

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

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

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

See Also "IOPEN", "IGETLUINFO", "IGETLU"

IGETONERROR

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IONERROR", "IGETERRNO", "IGETERRSTR", "ICAUSEERR"

IGETONINTR

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IONINTR", "IWAITHDLR", "IINTROFF", "IINTRON"

IGETONSRQ

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IONSRQ", "IWAITHDLR", "IINTROFF", "IINTRON"

IGETSESSTYPE

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

C Syntax

```
#include <sicl.h>

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

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

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

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

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

See Also "IOPEN"

IGETTERMCHR

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "ITERMCHR"

IGETTIMEOUT

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "ITIMEOUT"

IGPIBATNCTL

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

C Syntax

```
#include <sicl.h>

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

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

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

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

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

See Also "IGPIBSENDCMD", "IGPIBRENCTL", "IWRITE"

IGPIBBUSADDR

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IGPIBBUSSTATUS"

IGPIBBUSSTATUS

Supported sessions: . . . . . interface

C Syntax

```
#include <sicl.h>

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

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

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

`I_GPIB_BUS_ADDR`	Returns the bus address (0-30) of this interface on the GPIB bus.
`I_GPIB_BUS_LINES`	Returns the state of various GPIB lines. The result is a bit mask with the following bits being significant (bit 0 is the least-significant-bit): Bit 0: 1 if SRQ line is asserted. Bit 1: 1 if NDAC line is asserted. Bit 2: 1 if ATN line is asserted. Bit 3: 1 if DAV line is asserted. Bit 4: 1 if NRFD line is asserted. Bit 5: 1 if EOI line is asserted. Bit 6: 1 if IFC line is asserted. Bit 7: 1 if REN line is asserted. Bit 8: 1 if in REMote state. Bit 9: 1 if in LLO (local lockout) mode. Bit 10: 1 if currently the active controller. Bit 11: 1 if addressed to talk. Bit 12: 1 if addressed to listen.

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

See Also "IGPIBPASSCTL", "IGPIBSENDCMD"

IGPIBGETT1DELAY

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IGPIBSETT1DELAY"

IGPIBLLO

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IREMOTE", "ILOCAL"

IGPIBPASSCTL

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IONINTR", "ISETINTR"

IGPIBPPOLL

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IGPIBPPOLLCONFIG", "IGPIBPPOLLRESP"

IGPIBPPOLLCONFIG

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

C Syntax

```
#include <sicl.h>

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

Description

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

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

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

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

Return Value

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

See Also

"IGPIBPPOLL", "IGPIBPPOLLRESP"

IGPIBPPOLLRESP

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IGPIBPPOLL", "IGPIBPPOLLCONFIG"

IGPIBRENCTL

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IGPIBATNCTL"

IGPIBSENDCMD

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

C Syntax

```
#include <sicl.h>

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

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

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

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

See Also "IGPIBATNCTL", "IWRITE"

IGPIBSETT1DELAY

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

C Syntax

```
#include <sicl.h>

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

Description This function sets the t1 delay on the GPIB interface associated with session *id*. The value is the time of t1 delay in nanoseconds, and should be no less than `I_GPIB_T1DELAY_MIN` or no greater than `I_GPIB_T1DELAY_MAX`.

Note that most GPIB interfaces only support a small number of t1 delays, so the actual value used by the interface could be different than that specified in the `igpibsett1delay` function. You can find out the actual value used by calling the `igpibgett1delay` function.

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

See Also "IGPIBGETT1DELAY"

IGPIOCTRL

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

C Syntax

```
#include <sicl.h>

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

Note GPIO is *not* supported over LAN.

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

`I_GPIO_AUTO_HDSK`	If the *setting* parameter is non-zero, then the interface uses auto-handshake mode (the default). This gives the best performance for `iread` and iwrite operations. If the *setting* parameter is zero (`0`), then auto-handshake mode is canceled. This is *required* for programs that implement their own handshake using `I_GPIO_SET_PCTL`.
`I_GPIO_AUX`	The *setting* parameter is a mask containing the state of all auxiliary control lines. A 1 bit asserts the corresponding line; a 0 (zero) bit clears the corresponding line. When configured in Enhanced Mode, the HP/Agilent E2074/5 interface has 16 auxiliary control lines. In HP 98622 Compatibility Mode, it has none. Attempting to use `I_GPIO_AUX` in HP 98622 Compatibility Mode results in the error: `Operation not supported`.
`I_GPIO_CHK_PSTS`	If the *setting* parameter is non-zero, then the PSTS line is checked before each block of data is transferred. If the *setting* parameter is zero (`0`), then the PSTS line is ignored during data transfers. If the PSTS line is checked and false, SICL reports the error: `Device not active or available`.

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

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

`I_GPIO_CTRL_CTL0` The CTL0 line.

`I_GPIO_CTRL_CTL1` The CTL1 line.

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

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

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

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

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

Changes made by this function can remain in the interface hardware after your program ends. The *setting* remains until the computer is rebooted.

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

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

Changes made by this function can remain in the interface hardware after your program ends. The *setting* remains until the computer is rebooted.

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

Changes made by this function can remain in the interface hardware after your program ends. The *setting* remains until the computer is rebooted.

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

See Also "IGPIOSTAT", "IGPIOSETWIDTH"

IGPIOGETWIDTH

Supported sessions: . . . . . interface

C Syntax

```
#include <sicl.h>

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

Note GPIO is *not* supported over LAN.

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

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

See Also "IGPIOSETWIDTH"

IGPIOSETWIDTH

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

C Syntax

```
#include <sicl.h>

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

Note GPIO is *not* supported over LAN.

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

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

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

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

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

/* Using the GPIO interface in 16-bit mode */
igpiosetwidth(id, 16);
/* This call is platform-independent */
iwrite(id, words, 10, ... );
```

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

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

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

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

See Also "IGPIOGETWIDTH"

IGPIOSTAT

Supported sessions: . . . . . interface

C Syntax

```
#include <sicl.h>

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

Note GPIO is *not* supported over LAN.

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

`I_GPIO_CTRL`	The *result* is a mask representing the state of all control lines. The interface has two control lines, so only the two least-significant bits have meaning for that interface. These can be represented by the following. All other bits in the *result* mask are 0 (zero). `I_GPIO_CTRL_CTL0` The CTL0 line. `I_GPIO_CTRL_CTL1` The CTL1 line.
`I_GPIO_DATA`	The *result* is a mask representing the state of all data input latches. The interface can have either 8 or 16 data in lines, depending on the setting specified by `igpiosetwidth`. Note that this function reads the data lines asynchronously, without any type of handshake. It is intended for programs that implement their own handshake explicitly. An `igpiostat` function from one process will proceed even if another process has a lock on the interface. Ordinarily, this does not alter or disrupt any hardware states. Reading the data in lines is one exception. A data read causes an “input” indication on the I/O line (pin 20). In rare cases, that change might be unexpected, or undesirable, to the session that owns the lock.
`I_GPIO_INFO`	The *result* is a mask representing the following information about the device and the interface:
`I_GPIO_PSTS`	State of the PSTS line.
`I_GPIO_EIR`	State of the EIR line.
`I_GPIO_READY`	True if ready for a handshake. (Exact meaning depends on the current handshake mode.)
`I_GPIO_AUTO_HDSK`	True if auto-handshake mode is enabled. False if auto-handshake mode is disabled.

`I_GPIO_CHK_PSTS`	True if the PSTS line is to be checked before each block of data is transferred. False if PSTS is to be ignored during data transfers.
`I_GPIO_ENH_MODE`	True if the data ports are configured in Enhanced (bi-directional) Mode. False if the ports are configured in HP 98622 Compatibility Mode.
`I_GPIO_READ_EOI`	The *result* is the value of the current END pattern being used for read operations. If the *result* is `I_GPIO_EOI_NONE`, then no END pattern matching is being used. Any other *result* is the value of the END pattern.
`I_GPIO_STAT`	The *result* is a mask representing the state of all status lines. The interface has two status lines, so only the two least-significant bits have meaning for that interface. These can be represented by the following. All other bits in the *result* mask are 0 (zero). `I_GPIO_STAT_STI0`The STI0 line. `I_GPIO_STAT_STI1`The STI1 line.

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

See Also "IGPIOCTRL", "IGPIOSETWIDTH"

IHINT

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

C Syntax

```
#include <sicl.h>

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

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

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

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

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

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

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

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

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

IINTROFF

C Syntax

```
#include <sicl.h>

int iintroff ();
```

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

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

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

Default is on.

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

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

IINTRON

C Syntax

```
#include <sicl.h>

int iintron ();
```

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

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

Default is on.

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

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

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

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

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

ILANGETTIMEOUT

Supported sessions: . . . . . interface

C Syntax

```
#include <sicl.h>

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

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

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

See Also "ILANTIMEOUT" and Chapter 8 - Using SICL with LAN

ILANTIMEOUT

Supported sessions: . . . . . interface

C Syntax

```
#include <sicl.h>

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

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

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

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

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

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

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

Using the No-Wait Value

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

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

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

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

  Note that the server will send a reply to the write, even though the client will simply discard it. There is no way to directly determine the success or failure of the write, although a subsequent, functioning read call can be a good sign.
- Reset the client side timeout to a reasonable value for your network by calling `ilantimeout` again with a value sufficiently large enough to allow a read reply to be received. It is recommended that you use a value which provides some margin for error. Note that the timeout specified to `ilantimeout` is in milliseconds (rounded up to the nearest second).
- Make the blocking `iread` call (or formatted I/O call that will read data). Since `ilantimeout` has been set to a value other than `1` (preferably not `0`), the `iread` call will wait for a response from the server for the specified time (rounded up to the nearest second).

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

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

See Also ILANGETTIMEOUT and Chapter 8 - Using SICL with LAN

ILOCAL

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IREMOTE", and the interface-specific chapter of this manual for details of implementation.

ILOCK

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

C Syntax

```
#include <sicl.h>

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

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

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

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

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

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

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

Locking an **interface** (from an interface session) restricts other device and interface sessions from accessing this interface.

Locking a **device** restricts other device sessions from accessing this device; however, other interface sessions may continue to use this interface.

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

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

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

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

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

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

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

See Also "IUNLOCK", "ISETLOCKWAIT", "IGETLOCKWAIT"

IMAP

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

Note Not recommended for new program development. Use IMAPX instead.

C Syntax

```
#include <sicl.h>

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

Note Not supported over LAN.

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

The *id* argument specifies a VXI interface or device. The *pagestart* argument indicates the page number within the given memory space where the memory mapping starts. The *pagecnt* argument indicates how many pages to use.

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

`I_MAP_A16`	Map in VXI A16 address space (64 Kbyte pages).
`I_MAP_A24`	Map in VXI A24 address space (64 Kbyte pages).
`I_MAP_A32`	Map in VXI A32 address space (64 Kbyte pages).
`I_MAP_VXIDEV`	Map in VXI device registers. (Device session only, 64 bytes.)

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

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

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

The *suggested* argument, if non-NULL, contains a suggested address to begin mapping memory. However, the function may not always use this suggested address.

After memory is mapped, it may be accessed directly. Since this function returns a C pointer, you can also use C pointer arithmetic to manipulate the pointer and access memory directly. Note that accidentally accessing non-existent memory will cause bus errors. Refer to Chapter 6 of this manual for an example of trapping bus errors. Or see your operating system's programming information for help in trapping bus errors. You will probably find this information under the command `signal` in your operating system's manuals.

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

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

Return Value For C programs, this function returns a zero (0) if an error occurs or a non-zero number if successful. This non-zero number is the address to begin mapping memory.

See Also "IUNMAP", "IMAPINFO"

IMAPX

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

C Syntax

```
#include <sicl.h>

unsigned long imapx (id, mapspace, pagestart, pagecnt);
   INST id;
   int mapspace;
   unsigned int pagestart;
   unsigned int pagecnt;
```

Note Not supported over LAN.

Description The `imapx` function returns an unsigned long number, used in other functions, that maps a memory space into your process space. The SICL `ipeek?x` and `ipoke?x` functions can then be used to read and write to VXI address space.

The *id* argument specifies a VXI interface or device. The *pagestart* argument indicates the page number within the given memory space where the memory mapping starts. The *pagecnt* argument indicates how many pages to use.

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

`I_MAP_A16`	Map in VXI A16 address space (64 Kbyte pages).
`I_MAP_A24`	Map in VXI A24 address space (64 Kbyte pages).
`I_MAP_A32`	Map in VXI A32 address space (64 Kbyte pages).
`I_MAP_VXIDEV`	Map in VXI device registers. (Device session only, 64 bytes.)

`I_MAP_EXTEND`	Map in VXI Device Extended Memory address space in A24 or A32 address space. See individual device manuals for details regarding extended memory address space. (Device session only.)
`I_MAP_SHARED`	Map in VXI A24/A32 memory that is physically located on this device (sometimes called local shared memory). If the hardware supports it (that is, the local shared VXI memory is dual-ported), this map should be through the local system bus and not through the VXI memory. This mapping mechanism provides an alternate way of accessing local VXI memory without having to go through the normal VXI memory system. The value of *pagestart* is the offset (in 64 Kbyte pages) into the shared memory. The value of *pagecnt* is the amount of memory (in 64 Kbyte pages) to map.

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

Depending on what *iderefptr* returns, memory may be accessed directly. Since this function returns a C pointer, you can also use C pointer arithmetic to manipulate the pointer and access memory directly. Note that accidentally accessing non-existent memory will cause bus errors. Refer to Chapter 6 of this manual for an example of trapping bus errors. Or see your operating system's programming information for help in trapping bus errors. You will probably find this information under the command `signal` in your operating system's manuals.

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

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

Return Value For C programs, this function returns a zero (0) if an error occurs or a non-zero number if successful. This non-zero number is either a handle or the address to begin mapping memory. Use the `iderefptr` function to determine wheter the returned handle is a valid address or a handle.

See Also "IUNMAPX", "IMAPINFO", "IDEREFPTR"

IMAPINFO

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

C Syntax

```
#include <sicl.h>

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

Note Not supported over LAN.

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

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

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

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

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

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

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

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

See Also "IMAP", "IUNMAP"

IONERROR

C Syntax

```
#include <sicl.h>

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

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

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

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

Error handlers are called with the following arguments:

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

The *id* argument indicates the session that generated the error.The *error* argument indicates the error that occurred. See Chapter 9, Troubleshooting Your SICL Program, for a list of errors, for a complete description of the error codes.

Note The `INST` *id* that is passed to the error handler is the same `INST` *id* that was passed to the function that generated the error. Therefore, if an error occurred because of an invalid `INST` *id*, the `INST` *id* passed to the error

handler is also invalid. Also, if `iopen` generates an error before a session has been established, the error handler will be passed a zero (`0`) `INST` *id*.

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

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

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

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

Example for using *setjmp*/*longjmp*:

```
#include <sicl.h>

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

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

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

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

```
      if (igeterrno () ==...)
      ... etc ...;
   }
}
```

Or, using *try*/*recover*/*escape*:

```
#include <sicl.h>

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

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

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

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

See Also "IGETONERROR", "IGETERRNO", "IGETERRSTR", "ICAUSEERR"

IONINTR

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

C Syntax

```
#include <sicl.h>

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

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

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

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

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

Where:

id — The `INST` that refers to the session that installed the interrupt handler.

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

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

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

Valid *reason* values for all device sessions are:

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

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

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

See Also "ISETINTR", "IGETONINTR", "IWAITHDLR", "IINTROFF", "IINTRON", and the section titled "Asynchronous Events and HP-UX Signals" in Chapter 3 of this manual for protecting I/O calls against interrupts.

IONSRQ

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

C Syntax

```
#include <sicl.h>

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

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

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

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

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

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

See Also "IGETONSRQ", "IWAITHDLR", "IINTROFF", "IINTRON", "IREADSTB"

IOPEN

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

C Syntax

```
#include <sicl.h>

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

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

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

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

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

Note If an error handler has been installed (see `ionerror`), and an `iopen` generates an error before a session has been established, the handler will be called with the session identifier set to zero (`0`). Caution must be used if using the session identifier in an error handler.

Also, it is possible for an `iopen` to succeed on a device that does not exist. In this case, other functions (such as `iread`) will fail with a nonexistent device error.

Creating A Device Session

To create a device session, specify a particular interface name followed by the device's address in the *addr* parameter. For more information on addressing devices, see Chapter 3 - Using SICL.

C example:

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

Creating An Interface Session

To create an interface session, specify a particular interface in the *addr* parameter. For more information on addressing interfaces, see Chapter 3 - Using SICL.

C example:

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

Creating A Commander Session

To create a commander session, use the keyword `cmdr` in the *addr* parameter. For more information on commander sessions, see Chapter 3 - Using SICL.

C example:

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

Return Value

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

See Also

"ICLOSE"

IPEEK

Note Not recommended for new program development. Use IPEEKX8, IPEEKX16, IPEEKX32 instead.

C Syntax

```
#include <sicl.h>

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

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

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

Note Not supported over LAN.

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

Note The `iwpeek` and `ilpeek` functions perform byte swapping (if necessary) so that VXI memory accesses follow correct VXI byte ordering.
Also, if a bus error occurs, unexpected results may occur.

See Also "IPOKE", "IMAP"

IPEEKX8, IPEEKX16, IPEEKX32

C Syntax

```
#include <sicl.h>

int ipeekx8 (id, handle, offset, *value);
   INST id;
   unsigned long handle;
   unsigned long offset;
   unsigned char *value;

int ipeekx16 (id, handle, offset, *value);
   INST id;
   unsigned long handle;
   unsigned long offset;
   unsigned short *value

int ipeekx32 (id, handle, offset, *value);
   INST id;
   unsigned long handle;
   unsigned long offset;
   unsigned long *value))
```

Note Not supported over LAN.

Description The `ipeekx8, ipeekx16,` and `ipeekx32` functions read the values stored at *handle* and *offset* from memory and returns the value from that address. These functions are generally used in conjunction with the SICL `imapx` function to read data from VXI address space.

Note The `ipeekx8` and `ipeekx16` functions perform byte swapping (if necessary) so that VXI memory accesses follow correct VXI byte ordering. Also, if a bus error occurs, unexpected results may occur.

See Also "IPOKEX8, IPOKEX16, IPOKEX32", "IMAPX"

IPOKE

Note Not recommended for new program development. Use IPOKEX8, IPOKEX16, IPOKEX32 instead.

C Syntax

```
#include <sicl.h>

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

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

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

Note Not supported over LAN.

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

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

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

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

See Also "IPEEK", "IMAP"

IPOKEX8, IPOKEX16, IPOKEX32

C Syntax

```
#include <sicl.h>

int ipokex8 (id, handle, offset, value);
   INST id;
   unsigned long handle;
   unsigned long offset;
   unsigned char value;

int ipokex16 (id, handle, offset, value);
   INST id;
   unsigned long handle;
   unsigned long offset;
   unsigned short value;

int ipokex8 (id, handle, offset, value);
   INST id;
   unsigned long handle;
   unsigned long offset;
   unsigned long value;
```

Note Not supported over LAN.

Description The `ipokex8, ipokex16, and ipokex32` functions write to memory. The functions are generally used in conjunction with the SICL `imapx` function to write to VXI address space.

The *handle* is a valid memory address, *offset* is a valid memory offset. The *val* is a valid data value.

Note The `ipokex16` and `ipokex32` functions perform byte swapping (if necessary) so that VXI memory accesses follow correct VXI byte ordering. Also, if a bus error occurs, unexpected results may occur.

See Also "IPEEKX8, IPEEKX16, IPEEKX32", "IMAPX"

IPOPFIFO

C Syntax

```
#include <sicl.h>

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

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

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

Note Not supported over LAN.

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

The *id*, although specified, is normally ignored except to determine an interface-specific transfer mechanism such as DMA. To prevent using an interface-specific mechanism, pass a zero (`0`) in this parameter. The *dest* argument is the starting memory address for the destination data. The *fifo* argument is the memory address for the source FIFO register data. The *cnt* argument is the number of transfers (bytes, words, or longwords) to perform. The *swap* argument is the byte swapping flag. If *swap* is zero, no swapping

occurs. If *swap* is non-zero, the function swaps bytes (if necessary) to change byte ordering from the internal format of the controller to/from the VXI (big-endian) byte ordering.

Note If a bus error occurs, unexpected results may occur.

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

See Also "IPEEK", "IPOKE", "IPUSHFIFO", "IMAP"

IPRINTF

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

C Syntax

```
#include <sicl.h>

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

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

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

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

Re-addressing occurs under the following circumstances:

- After the internal buffer fills. (See isetbuf.)
- When a %C is found in the *format* string.

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

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

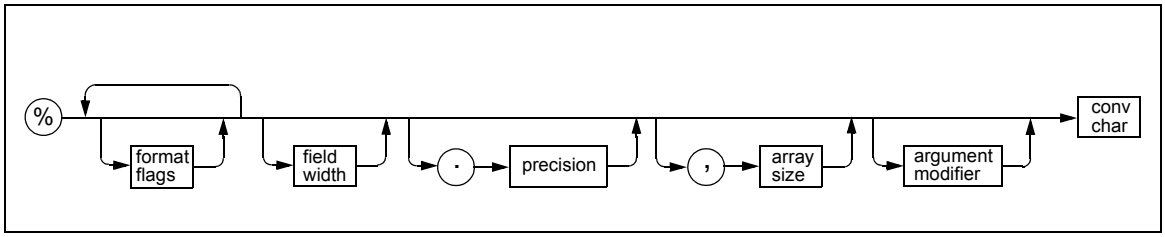
Special Characters for C/C++

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

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

Format Conversion Commands

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



The modifiers in a conversion command are:

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

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

. precision

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

, array size

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

argument modifier

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

conv char

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

Examples of Format Conversion Commands

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

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

List of *format flags*

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

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

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

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

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

`@Q`	Convert to an IEEE 488.2 format octal number in the form `#Qxxxx`. Valid only for `%d` and `%f`. Note that `%f` values will be truncated to the integer value.
`@B`	Convert to an IEEE 488.2 format binary number in the form `#Bxxxx`. Valid only for `%d` and `%f`. Note that `%f` values will be truncated to the integer value.
`-`	Left justify the result.
`+`	Prefix the result with a sign (`+` or `-`) if the output is a signed type.
`space`	Prefix the result with a blank ( ) if the output is signed and positive. Ignored if both blank and `+` are specified.
`#`	Use alternate form. For the o conversion, it prints a leading zero. For `x` or `X`, a non-zero will have `0x` or `0X` as a prefix. For `e`, `E`, `f`, `g`, and `G`, the result will always have one digit on the right of the decimal point.
`0`	Will cause the left pad character to be a zero (`0`) for all numeric conversion types.

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

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

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

`b` In C/C++, corresponding *arg* is a pointer to an arbitrary block of data. The data is sent as IEEE 488.2 Definite Length Arbitrary Block Response Data. The field width must be present and will specify the number of elements in the data block. An asterisk (`*`) can be used in place of the integer, which indicates that two *arg*s are used. The first is a long used to specify the number of elements. The second is the pointer to the data block. No byte swapping is performed.

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

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

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

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

`B` Same as `b` in C/C++, except that the data block is sent as IEEE 488.2 Indefinite Length Arbitrary Block Response Data. Note that this format involves sending a newline with an END indicator on the last byte of the data block.

`c` In C/C++, corresponding *arg* is a character.

`C` In C/C++, corresponding *arg* is a character. Send with END indicator.

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

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

`S` In C/C++, corresponding *arg* is a pointer to a null-terminated string that is sent as an IEEE 488.2 string response data block. An IEEE 488.2 string response data block consists of a leading double quote (") followed by non-double quote characters and terminated with a double quote.

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

`i`	Corresponding *arg* is an integer. Same as `d` except that the six flag modifier formatting options: NR1 - `@1`, NR2 - `@2`, NR3 - `@3`, `@H`, `@Q`, or `@B` are ignored.
`o,u,x,X`	Corresponding *arg* will be treated as an unsigned integer. The argument is converted to an unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal (`x`,`X`). The letters `abcdef` are used with `x`, and the letters `ABCDEF` are used with `X`. The precision specifies the minimum number of characters to appear. If the value can be represented with fewer than precision digits, leading zeros are added. If the precision is set to zero and the value is zero, no characters are printed.
`e,E`	Corresponding *arg* is a double in C/C++. The argument is converted to exponential format (that is, `[-]d.dddde+/-dd`). The precision specifies the number of digits to the right of the decimal point. If no precision is specified, then six digits will be converted. The letter e will be used with `e` and the letter E will be used with `E`.
`g,G`	Corresponding *arg* is a double in C/C++. The argument is converted to exponential (e with `g`, or E with `G`) or floating point format depending on the value of the *arg* and the precision. The exponential style will be used if the resulting exponent is less than -4 or greater than the precision; otherwise it will be printed as a float.
`n`	Corresponding *arg* is a pointer to an integer in C/C++. The number of bytes written to the device for the entire `iprintf` call is written to the *arg*. No argument is converted.
`F`	Corresponding *arg* is a pointer to a FILE descriptor. The data will be read from the file that the FILE descriptor points to and written to the device. The FILE descriptor must be opened for reading. No flags or modifiers are allowed with this conversion character.

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

Buffers and Errors

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

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

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

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

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

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

See Also

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

IPROMPTF

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

C Syntax

```
#include <sicl.h>

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

Description

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

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

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

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

See Also

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

IPUSHFIFO

C Syntax

```
#include <sicl.h>

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

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

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

Note Not supported over LAN.

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

The *id*, although specified, is normally ignored except to determine an interface-specific transfer mechanism such as DMA. To prevent using an interface-specific mechanism, pass a zero (`0`) in this parameter. The *src* argument is the starting memory address for the source data. The *fifo* argument is the memory address for the destination FIFO register data. The *cnt* argument is the number of transfers (bytes, words, or longwords) to perform. The *swap* argument is the byte swapping flag. If *swap* is zero, no

swapping occurs. If *swap* is non-zero the function swaps bytes (if necessary) to change byte ordering from the internal format of the controller to/from the VXI (big-endian) byte ordering.

Note If a bus error occurs, unexpected results may occur.

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

See Also "IPOPFIFO", "IPOKE", "IPEEK", "IMAP"

IREAD

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

C Syntax

```
#include <sicl.h>

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

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

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

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

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

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

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

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

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

See Also "IWRITE", "ITERMCHR", "IFREAD", "IFWRITE"

IREADSTB

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "IONSRQ", "ISETSTB"

IREMOTE

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

C Syntax

```
#include <sicl.h>

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

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

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

See Also "ILOCAL", and the interface-specific chapter in this manual for details of implementation.

ISCANF

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

C Syntax

```
#include <sicl.h>

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

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

The *format* string contains:

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

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

Notes on Using `iscanf`

- Using `itermchr` with `iscanf`:

  The `iscanf` function only terminates reading on an END indicator or the termination character specified by `itermchar`.

- Using `iscanf` with Certain Instruments:

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

- Buffer Management with iscanf:

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

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

  With:

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

  There are four ways to get the desired results:

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

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

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

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

-- Do explicit calls to `iflush` to flush the read buffer.

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

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

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

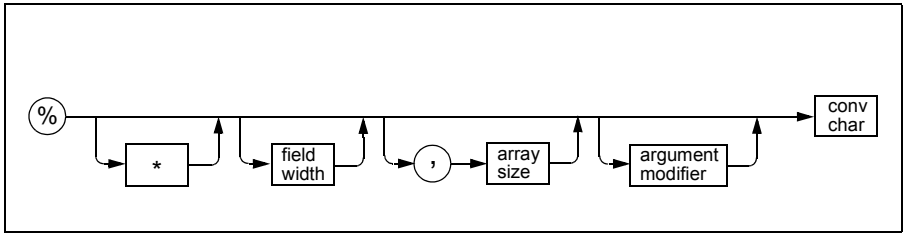
White-Space Characters for C/C++

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

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

Format Conversion Commands

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



The modifiers in a conversion command are:

*	An optional, assignment suppression character (`*`). This provides a way to describe an input field to be skipped. An input field is defined as a string of non-white-space characters that extends either to the next inappropriate character, or until the *field width* (if specified) is exhausted.
field width	An optional integer representing the *field width*. In C/C++, if a pound sign (`#`) appears instead of the integer, then the next *arg* is a pointer to the *field width*. This *arg* is a pointer to an integer for `%c`, `%s`, `%t`, and `%S`. This *arg* is a pointer to a long for `%b`. The field width is not allowed for `%d` or `%f`.
, array size	An optional comma operator is an integer preceded by a comma. It reads a list of comma-separated numbers. The comma operator is in the form of `,dd`, where `dd` is the number of array elements to read. In C/C++, a pound sign (`#`) can be substituted for the number, in which case the next argument is a pointer to an integer that is the number of elements in the array.

	The function will set this to the number of elements read. This operator is only valid with the conversion characters `d` and `f`. The argument must be an array of the type specified.
argument modifier	The meaning of the optional argument modifiers `h`, `l`, `w`, `z`, and `Z` is dependent on the conversion character.
conv char	A conversion character is a character that specifies the type of arg and the conversion to be applied. This is the only required element of a conversion command. See the following subsection, "List of conv chars" for the specific conversion characters you may use.

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

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

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

Examples of Format Conversion Commands

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

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

`%10c`	`onestring`	field width
`%10t`	`two strings`	field width (10 chars read into 1 arg)

List of *conv chars* The *conv chars* (conversion characters) are:

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

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

`f` Corresponding *arg* must be a pointer to a float in C/C++. The library reads characters until an entire number is read. It will convert IEEE 488.2 HEX, OCT, BIN, and NRf format numbers. If the `l` (ell) argument modifier is used, the argument must be a pointer to a double for C/C++. If the `L` argument modifier is used, the argument must be a pointer to a long double for C/C++ .

`e,g` Corresponding *arg* must be a pointer to a float for C/C++. The library reads characters until an entire number is read. If the `l` (ell) argument modifier is used, the argument must be a pointer to a double for C/C++. If the `L` argument modifier is used, the argument must be a pointer to a long double for C/C++.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`n`	Corresponding *arg* is a pointer to an integer for C/C++. The number of bytes currently converted from the device is placed into the arg. No argument is converted.
`F`	Corresponding *arg* is a pointer to a FILE descriptor. The input data read from the device is written to the file referred to by the FILE descriptor until the END indicator is received. The file must be opened for writing. No other modifiers or flags are valid with this conversion character.

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

`d`	IEEE 488.2 HEX, OCT, BIN, and NRf formats (for example, `#HA`, `#Q12`, `#B1010`, `10`, `10.00`, and `1.00E+01`).
`f`	IEEE 488.2 HEX, OCT, BIN, and NRf formats (for example, `#HA`, `#Q12`, `#B1010`, `10`, `10.00`, and `1.00E+01`).
`i`	Integer. Data with a leading 0 will be converted as octal; data with leading `0x` or `0X` will be converted as hexadecimal.
`u`	Unsigned integer. Same as `i` except value is unsigned.
`o`	Unsigned integer. Data will be converted as octal.
`x,X`	Unsigned integer. Data will be converted as hexadecimal.
`e,g`	Floating. Integers, floating point, and exponential numbers will be converted into floating point numbers (default is float).

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

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

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

ISERIALBREAK

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

C Syntax

```
#include <sicl.h>

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

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

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

ISERIALCTRL

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

C Syntax

```
#include <sicl.h>

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

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

`I_SERIAL_BAUD`	The *setting* parameter will be the new speed of the interface. The value should be a valid baud rate for the interface (for example, 300, 1200, 9600). The baud rate is represented as an unsigned long integer, in bits per second. If the value is not a recognizable baud rate, an err_param error is returned. The following are the supported baud rates: 50, 110, 300, 600, 1200, 2400, 4800, 7200, 9600, 19200, 38400, and 57600.
`I_SERIAL_PARITY`	The following values are acceptable values for *setting*: `I_SERIAL_PAR_EVEN` - Even parity `I_SERIAL_PAR_ODD` - Odd parity `I_SERIAL_PAR_NONE` - No parity bit is used `I_SERIAL_PAR_MARK` - Parity is always one `I_SERIAL_PAR_SPACE` - Parity always zero
`I_SERIAL_STOP`	The following are acceptable values for *setting*: `I_SERIAL_STOP_1` - 1 stop bit `I_SERIAL_STOP_2` - 2 stop bits

`I_SERIAL_WIDTH`	The following are acceptable values for *setting*: `I_SERIAL_CHAR_5` - 5 bit characters `I_SERIAL_CHAR_6` - 6 bit characters `I_SERIAL_CHAR_7` - 7 bit characters `I_SERIAL_CHAR_8` - 8 bit characters
`I_SERIAL_READ_BUFSZ`	This is used to set the size of the read buffer. The *setting* parameter is used as the size of buffer to use. This value must be in the range of 1 and 32767.
`I_SERIAL_DUPLEX`	The following are acceptable values for *setting*: `I_SERIAL_DUPLEX_FULL` - Use full duplex `I_SERIAL_DUPLEX_HALF` - Use half duplex
`I_SERIAL_FLOW_CTRL`	The *setting* parameter must be set to one of the following values. If no flow control is to be used, set *setting* to zero (`0`). The following are the supported types of flow control: `I_SERIAL_FLOW_NONE` - No handshaking `I_SERIAL_FLOW_XON` - Software handshake `I_SERIAL_FLOW_RTS_CTS` - Hardware handshake `I_SERIAL_FLOW_DTR_DSR` - Hardware handshake
`I_SERIAL_READ_EOI`	Used to set the type of END Indicator to use for reads. In order for `iscanf` to work as specified, data must be terminated with an END indicator. The RS-232 interface has no standard way of doing this. SICL gives you two different methods of indicating EOI.

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

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

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

`I_SERIAL_WRITE_EOI`	The *setting* parameter will contain the value of the type of END Indicator to use for writes. The following are valid values to use: • `I_SERIAL_EOI_NONE` - No EOI indicator. This is the default for `I_SERIAL_WRITE` (`iprintf`). • `I_SERIAL_EOI_BIT8` - Use the eighth bit of the data to indicate EOI. On the last byte, the eighth bit will be masked off, and the result will be placed into the buffer.
`I_SERIAL_RESET`	This will reset the serial interface. The following actions will occur: any pending writes will be aborted, the data in the input buffer will be discarded, and any error conditions will be reset. This differs from `iclear` in that no BREAK will be sent.

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

See Also "ISERIALSTAT"

ISERIALMCLCTRL

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

C Syntax

```
#include <sicl.h>

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

Description The `iserialmclctrl` function is used to control the Modem Control Lines. The *sline* parameter sends one of the following values:

`I_SERIAL_RTS` Ready To Send line
`I_SERIAL_DTR` Data Terminal Ready line

If the *state* value is non-zero, the Modem Control Line will be asserted; otherwise it will be cleared.

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

See Also "ISERIALMCLSTAT", "IONINTR", "ISETINTR"

ISERIALMCLSTAT

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

C Syntax

```
#include <sicl.h>

int iserialmclstat (id, sline, state);
INST id;
int sline;
int *state;
```

Description The `iserialmclstat` function is used to determine the current state of the Modem Control Lines. The *sline* parameter sends one of the following values:

`I_SERIAL_RTS` Ready To Send line
`I_SERIAL_DTR` Data Terminal Ready line

If the value returned in *state* is non-zero, the Modem Control Line is asserted; otherwise it is clear.

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

See Also "ISERIALMCLCTRL"

ISERIALSTAT

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

C Syntax

```
#include <sicl.h>

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

Description The `iserialstat` function is used to find the status of the serial interface. This function takes one of the following values passed in *request* and returns the status in the *result* parameter:

`I_SERIAL_BAUD` — The *result* parameter will be set to the speed of the interface.

`I_SERIAL_PARITY` — The *result* parameter will be set to one of the following values:

- `I_SERIAL_PAR_EVEN` - Even parity
- `I_SERIAL_PAR_ODD` - Odd parity
- `I_SERIAL_PAR_NONE` - No parity bit
- `I_SERIAL_PAR_MARK` - Parity always one
- `I_SERIAL_PAR_SPACE` - Parity always zero

`I_SERIAL_STOP` — The *result* parameter will be set to one of the following values:

- `I_SERIAL_STOP_1` - 1 stop bits
- `I_SERIAL_STOP_2` - 2 stop bits

`I_SERIAL_WIDTH` — The *result* parameter will be set to one of the following values:

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

`I_SERIAL_DUPLEX`

The *result* parameter will be set to one of the following values:

`I_SERIAL_DUPLEX_FULL` Use full duplex

`I_SERIAL_DUPLEX_HALF` Use half duplex

`I_SERIAL_MSL`

- The *result* parameter will be set to the bit wise OR of all of the Modem Status Lines that are currently being asserted. The value of the *result* parameter will be the logical OR of all of the serial lines currently being asserted. The serial lines are both the Modem Control Lines and the Modem Status Lines. The following are the supported serial lines:

- `I_SERIAL_DCD` - Data Carrier Detect.
- `I_SERIAL_DSR` - Data Set Ready.
- `I_SERIAL_CTS` - Clear To Send.
- `I_SERIAL_RI` - Ring Indicator.
- `I_SERIAL_TERI` - Trailing Edge of RI.
- `I_SERIAL_D_DCD` - The DCD line has changed since the last time this status has been checked.
- `I_SERIAL_D_DSR` - The DSR line has changed since the last time this status has been checked.
- `I_SERIAL_D_CTS` - The CTS line has changed since the last time this status has been checked.

`I_SERIAL_STAT`	This is a read destructive status. That means reading this request resets the condition. The *result* parameter will be set the bit wise OR of the following conditions: • `I_SERIAL_DAV` - Data is available. • `I_SERIAL_PARITY` - Parity error has occurred since the last time the status was checked. • `I_SERIAL_OVERFLOW` - Overflow error has occurred since the last time the status was checked. • `I_SERIAL_FRAMING` - Framing error has occurred since the last time the status was checked. • `I_SERIAL_BREAK` - Break has been received since the last time the status was checked. • `I_SERIAL_TEMT` - Transmitter empty.
`I_SERIAL_READ_BUFSZ`	The *result* parameter will be set to the current size of the read buffer.
`I_SERIAL_READ_DAV`	The *result* parameter will be set to the current amount of data available for reading.
`I_SERIAL_FLOW_CTRL`	The *result* parameter will be set to the value of the current type of flow control that the interface is using. If no flow control is being used, *result* will be set to zero (`0`). The following are the supported types of flow control: `I_SERIAL_FLOW_NONE` - No handshaking `I_SERIAL_FLOW_XON` - Software handshake `I_SERIAL_FLOW_RTS_CTS` - Hardware handshake `I_SERIAL_FLOW_DTR_DSR` - Hardware handshake

`I_SERIAL_READ_EOI`	The *result* parameter will be set to the value of the current type of END indicator that is being used for reads. The following values can be returned: • `I_SERIAL_EOI_CHR`|(*n*) - A character is used to indicate EOI, where *n* is the character. These two values are logically OR-ed together. To find the value of the character, AND *result* with 0xff. The default is a `\n`. • `I_SERIAL_EOI_NONE` - No EOI indicator. This is the default for `I_SERIAL_READ` (`iscanf`). • `I_SERIAL_EOI_BIT8` - Use the eighth bit of the data to indicate EOI. This last byte will mask off this bit and use the rest for the data that is put in your buffer.
`I_SERIAL_WRITE_EOI`	The *result* parameter will be set to the value of the current type of END indicator that is being used for reads. The following values can be returned: • `I_SERIAL_EOI_NONE` - No EOI indicator. This is the default for `I_SERIAL_WRITE` (`iprintf`). • `I_SERIAL_EOI_BIT8` - Use the eighth bit of the data to indicate EOI. This last byte will mask off this bit and use the rest for the data that is put in your buffer.

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

See Also "ISERIALCTRL"

ISETBUF

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

C Syntax

```
#include <sicl.h>

int isetbuf (id, mask, size);
INST id;
int mask;
int size;
```

Description This function is used to set the size and actions of the read and/or write buffers of formatted I/O. The *mask* can be one or the bit-wise OR of both of the following flags:

`I_BUF_READ` Specifies the read buffer.
`I_BUF_WRITE` Specifies the write buffer.

The *size* argument specifies the size of the read or write buffer (or both) in bytes. Setting a size of zero (`0`) disables buffering. This means that for write buffers, each byte goes directly to the device. For read buffers, the driver reads each byte directly from the device.

Setting a size greater than zero creates a buffer of the specified size. For write buffers, the buffer flushes (writes to the device) whenever the buffer fills up and for each newline character in the format string. (However, note that the buffer is *not* flushed by newline characters in the argument list.) For read buffers, the buffer is never flushed (that is, it holds any leftover data for the next `iscanf`/`ipromptf` call). This is the default action.

Setting a size less than zero creates a buffer of the absolute value of the specified size. For write buffers, the buffer flushes (writes to the device) whenever the buffer fills up, for each newline character in the format string, or at the completion of every `iprintf` call. For read buffers, the buffer flushes (erases its contents) at the end of every `iscanf` (or `ipromptf`) function.

Note Calling `isetbuf` flushes any data in the buffer(s) specified in the *mask* parameter.

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

See Also "IPRINTF", "ISCANF", "IPROMPTF", "IFWRITE", "IFREAD", "IFLUSH", "ISETUBUF"

ISETDATA

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

C Syntax

```
#include <sicl.h>

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

Description The `isetdata` function stores a pointer to a data structure and associates it with a session (or `INST` *id*).

You can use these user-defined data structures to associate device-specific data with a session such as device name, configuration, instrument settings, and so forth.

You are responsible for the management of the buffer (that is, if the buffer needs to be allocated or deallocated, you must do it).

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

See Also "IGETDATA"

ISETINTR

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

C Syntax

```
#include <sicl.h>

int isetintr (id, intnum, secval);
INST id;
int intnum;
long secval;
```

Description The `isetintr` function is used to enable interrupt handling for a particular event. Installing an interrupt handler only allows you to receive enabled interrupts. By default, all interrupt events are disabled.

The *intnum* parameter specifies the possible causes for interrupts. A valid *intnum* value for *any* type of session is:

`I_INTR_OFF`	Turns off all interrupt conditions previously enabled with calls to `isetintr`.

A valid intnum value for *all* **device sessions** (except for GPIB and GPIO, which have no device-specific interrupts) is:

`I_INTR_*`	Individual interfaces may include other interface-interrupt conditions. See the following information on each interface for more details.

Valid *intnum* values for *all* **interface** sessions are:

`I_INTR_INTFACT`	Interrupt when the interface becomes active. Enable if *secval*!=0; disable if *secval*=0.
`I_INTR_INTFDEACT`	Interrupt when the interface becomes deactivated. Enable if *secval*!=0; disable if *secval*=0.

`I_INTR_TRIG`	Interrupt when a trigger occurs. The *secval* parameter contains a bit-mask specifying which triggers can cause an interrupt. See the `ixtrig` function's *which* parameter for a list of valid values.
`I_INTR_*`	Individual interfaces may include other interface-interrupt conditions. See the following information on each interface for more details.

Valid *intnum* values for *all* **commander sessions** (except RS-232 and GPIO, which do not support commander sessions) are:

`I_INTR_STB`	Interrupt when the commander reads the status byte from this controller. Enable if *secval*!=0; disable if *secval*=0.
`I_INTR_DEVCLR`	Interrupt when the commander sends a device clear to this controller (on the given interface). Enable if *secval*!=0; disable if *secval*=0.

Interrupts on GPIB

GPIB Device Session Interrupts. There are no device-specific interrupts for the GPIB interface.

GPIB Interface Session Interrupts. The interface-specific interrupt for the GPIB interface is:

`I_INTR_GPIB_IFC`	Interrupt when an interface clear occurs. Enable when *secval*!=0; disable when *secval*=0. This interrupt will be generated regardless of whether this interface is the system controller or not (that is, regardless of whether this interface generated the IFC, or another device on the interface generated the IFC).

The following are generic interrupts for the GPIB interface:

`I_INTR_INTFACT`	Interrupt occurs whenever this controller becomes the active controller.
`I_INTR_INTFDEACT`	Interrupt occurs whenever this controller passes control to another GPIB device. (For example, the `igpibpassctl` function has been called.)

GPIB Commander Session Interrupts. The following are commander-specific interrupts for GPIB:

`I_INTR_GPIB_PPOLLCONFIG`	This interrupt occurs whenever there is a change to the PPOLL configuration. This interrupt is enabled using `isetintr` by specifying a *secval* greater than 0. If *secval*=0, this interrupt is disabled.
`I_INTR_GPIB_REMLOC`	This interrupt occurs whenever a remote or local message is received and addressed to listen. This interrupt is enabled using `isetintr` by specifying a *secval* greater than 0. If *secval*=0, this interrupt is disabled.
`I_INTR_GPIB_GET`	This interrupt occurs whenever the GET message is received and addressed to listen. This interrupt is enabled using `isetintr` by specifying a *secval* greater than 0. If *secval*=0, this interrupt is disabled.

`I_INTR_GPIB_TLAC`	• This interrupt occurs whenever this device has been addressed to talk or untalk, or the device has been addressed to listen or unlisten. When the interrupt handler is called, the *secval* value is set to a bit mask. Bit 0 is for listen, and bit 1 is for talk. If: Bit 0 = 1, then this device is addressed to listen. • Bit 0 = 0, then this device is not addressed to listen. • Bit 1 = 1, then this device is addressed to talk. • Bit 1 = 0, then this device is not addressed to talk. This interrupt is enabled using *isetintr* by specifying a *secval* greater than 0. If *secval*=0, this interrupt is disabled.

Interrupts on GPIO

GPIO Device Session Interrupts. GPIO does not support device sessions. Therefore, there are no device session interrupts for GPIO.

GPIO Interface Session Interrupts. The interface-specific interrupts for the GPIO interface are:

`I_INTR_GPIO_EIR`	This interrupt occurs whenever the EIR line is asserted by the peripheral device. Enabled when *secval*!=0, disabled when *secval*=0.
`I_INTR_GPIO_RDY`	This interrupt occurs whenever the interface becomes ready for the next handshake. (The exact meaning of "ready" depends on the configured handshake mode.) Enabled when *secval*!=0, disabled when *secval*=0.

Note The GPIO interface is always active. Therefore, the interrupts for `I_INTR_INTFACT` and `I_INTR_INTFDEACT` will never occur.

GPIO Commander Session Interrupts. GPIO does not support commander sessions. Therefore, there are no commander session interrupts for GPIO.

Interrupts on RS-232 (Serial)

RS-232 Device Session Interrupts. The device-specific interrupt for the RS-232 interface is:

`I_INTR_SERIAL_DAV` This interrupt occurs whenever the receive buffer in the driver goes from the empty to the non-empty state.

RS-232 Interface Session Interrupts. The interface-specific interrupts for the RS-232 interface are:

`I_INTR_SERIAL_MSL` This interrupt occurs whenever one of the specified modem status lines changes states. The *secval* argument in `ionintr` is the logical OR of the Modem Status Lines to monitor. In the interrupt handler, the *sec* argument will be the logical OR of the MSL line(s) that caused the interrupt handler to be invoked.

Note that most implementations of the ring indicator interrupt only deliver the interrupt when the state goes from high to low (that is, a trailing edge). This differs from the other MSLs in that it's not simply just a state change that causes the interrupts.

The status lines that can cause this interrupt are DCD, CTS, DSR, and RI.

`I_INTR_SERIAL_BREAK`	This interrupt occurs whenever a BREAK is received.
`I_INTR_SERIAL_ERROR`	This interrupt occurs whenever a parity, overflow, or framing error happens. The *secval* argument in `ionintr` is the logical OR of one or more of the following values to enable the appropriate interrupt. In the interrupt handler, the *sec* argument will be the logical OR of these values that indicate which error(s) occurred: • `I_SERIAL_PARERR` - Parity Error • `I_SERIAL_OVERFLOW`- Buffer Overflow Error • `I_SERIAL_FRAMING` - Framing Error
`I_INTR_SERIAL_DAV`	This interrupt occurs whenever the receive buffer in the driver goes from the empty to the non-empty state.
`I_INTR_SERIAL_TEMT`	This interrupt occurs whenever the transmit buffer in the driver goes from the non-empty to the empty state.

The following are generic interrupts for the RS-232 interface:

`I_INTR_INTFACT`	This interrupt occurs when the Data Carrier Detect (DCD) line is asserted.
`I_INTR_INTFDEACT`	This interrupt occurs when the Data Carrier Detect (DCD) line is cleared.

RS-232 Commander Session Interrupts. RS-232 does not support commander sessions. Therefore, there are no commander session interrupts for RS-232.

Interrupts on VXI

VXI Device Session Interrupts. The device-specific interrupt for the VXI interface is:

`I_INTR_VXI_SIGNAL`	A specified device wrote to the VXI signal register (or a VME interrupt arrived from a VXI device that is in the servant list), and the signal was an event you defined. This interrupt is enabled using `isetintr` by specifying a *secval*!=0. If *secval*=0, then this is disabled. The value written into the signal register is returned in the *secval* parameter of the interrupt handler.

VXI Interface Session Interrupts. The following are interface-specific interrupts for the VXI interface:

`I_INTR_VXI_SYSRESET`	A VXI SYSRESET occurred. This interrupt is enabled using `isetintr` by specifying a *secval*!=0. If *secval*=0, then this is disabled.
`I_INTR_VXI_VME`	A VME interrupt occurred from a non-VXI device, or a VXI device that is not a servant of this interface. This interrupt is enabled using `isetintr` by specifying a *secval*!=0. If *secval*=0, then this is disabled.
`I_INTR_VXI_UKNSIG`	A write to the VXI signal register was performed by a device that is not a servant of this controller. This interrupt condition is enabled using `isetintr` by specifying a *secval*!=0. If *secval*=0, then this is disabled. The value written into the signal register is returned in the *secval* parameter of the interrupt handler.
`I_INTR_VXI_VMESYSFAIL`	The VME SYSFAIL line has been asserted.
`I_INTR_VME_IRQ1`	VME IRQ1 has been asserted.

`I_INTR_VME_IRQ2`	VME IRQ2 has been asserted.
`I_INTR_VME_IRQ3`	VME IRQ3 has been asserted.
`I_INTR_VME_IRQ4`	VME IRQ4 has been asserted.
`I_INTR_VME_IRQ5`	VME IRQ5 has been asserted.
`I_INTR_VME_IRQ6`	VME IRQ6 has been asserted.
`I_INTR_VME_IRQ7`	VME IRQ7 has been asserted.

The following are generic interrupts for the VXI interface:

`I_INTR_INTFACT`	This interrupt occurs whenever the interface receives a BNO (Begin Normal Operation) message.
`I_INTR_INTFDEACT`	This interrupt occurs whenever the interface receives an ANO (Abort Normal Operation) or ENO (End Normal Operation) message.

VXI Commander Session Interrupts. The commander-specific interrupt for VXI is:

`I_INTR_VXI_LLOCK`	A lock/clear lock word-serial command has arrived. This interrupt is enabled using `isetintr` by specifying a *secval*!=0. If *secval*=0, then this is disabled. If a lock occurred, the *secval* in the handler is passed a 1; if an unlock, the *secval* in the handler is passed 0.

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

See Also "IONINTR", "IGETONINTR", "IWAITHDLR", "IINTROFF", "IINTRON", "IXTRIG", and the section titled "Asynchronous Events and HP-UX Signals" in Chapter 3 for protecting I/O calls against interrupts.

ISETLOCKWAIT

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

C Syntax

```
#include <sicl.h>

int isetlockwait (id, flag);
INST id;
int flag;
```

Description The `isetlockwait` function determines whether library functions wait for a device to become unlocked or return an error when attempting to operate on a locked device. The error that is returned is `I_ERR_LOCKED`.

If *flag* is non-zero, then all operations on a device or interface locked by another session will wait for the lock to be removed. This is the default case.

If *flag* is zero (0), then all operations on a device or interface locked by another session will return an error (`I_ERR_LOCKED`). This will disable the timeout value set up by the `itimeout` function.

Note If a request is made that cannot be granted due to hardware constraints, the process will hang until the desired resources become available. To avoid this, use the `isetlockwait` command with the *flag* parameter set to 0, and thus generate an error instead of waiting for the resources to become available.

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

See Also "ILOCK", "IUNLOCK", "IGETLOCKWAIT"

ISETSTB

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

C Syntax

```
#include <sicl.h>

int isetstb (id, stb);
INST id;
unsigned char stb;
```

Description The `isetstb` function allows the status byte value for this controller to be changed. This function is only valid for commander sessions.

Bit 6 in the *stb* (status byte) has special meaning. If bit 6 is set, then an SRQ notification is given to the remote controller, if its identity is known. If bit 6 is not set, then the SRQ notification is canceled. The exact mechanism for sending the SRQ notification is dependent on the interface.

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

See Also "IREADSTB", "IONSRQ"

ISETUBUF

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

C Syntax

```
#include <sicl.h>

int isetubuf (id, mask, size, buf);
INST id;
int mask;
int size;
char *buf;
```

Description

The `isetubuf` function is used to supply the buffer(s) used for formatted I/O. With this function you can specify the size and the address of the formatted I/O buffer.

This function is used to set the size and actions of the read and/or write buffers of formatted I/O. The *mask* may be one, but NOT both of the following flags:

`I_BUF_READ`	Specifies the read buffer.
`I_BUF_WRITE`	Specifies the write buffer.

Setting a *size* greater than zero creates a buffer of the specified size. For write buffers, the buffer flushes (writes to the device) whenever the buffer fills up and for each newline character in the format string. For read buffers, the buffer is never flushed (that is, it holds any leftover data for the next `iscanf`/`ipromptf` call). This is the default action.

Setting a *size* less than zero creates a buffer of the absolute value of the specified size. For write buffers, the buffer flushes (writes to the device) whenever the buffer fills up, for each newline character in the format string, or at the completion of every `iprintf` call. For read buffers, the buffer flushes (erases its contents) at the end of every `iscanf` (or `ipromptf`) function.

Note Calling `isetubuf` flushes the buffer specified in the mask parameter.

Note Once a buffer is allocated to `isetubuf`, do not use the buffer for any other use. In addition, once a buffer is allocated to `isetubuf` (either for a read or write buffer), don't use the same buffer for any other session or for the opposite type of buffer on the same session (write or read, respectively).

In order to free a buffer allocated to a session, make a call to `isetbuf`, which will cause the user-defined buffer to be replaced by a system-defined buffer allocated for this session. The user-defined buffer may then be either re-used, or freed by the program.

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

See Also "IPRINTF", "ISCANF", "IPROMPTF", "IFWRITE", "IFREAD", "ISETBUF", "IFLUSH"

ISWAP

C Syntax

```
#include <sicl.h>

int iswap (addr, length, datasize);
int ibeswap (addr, length, datasize);
int ileswap (addr, length, datasize);
char *addr;
unsigned long length;
int datasize;
```

Description These functions provide an architecture-independent way of byte swapping data received from a remote device or data that is to be sent to a remote device. This data may be received/sent using the `iwrite`/`iread` calls, or the `ifwrite`/`ifread` calls.

The `iswap` function will always swap the data.

The `ibeswap` function assumes the data is in big-endian byte ordering (big-endian byte ordering is where the most significant byte of data is stored at the least significant address) and converts the data to whatever byte ordering is native on this controller's architecture. Or it takes the data that is byte ordered for this controller's architecture and converts the data to big-endian byte ordering. (Notice that these two conversions are identical.)

The `ileswap` function assumes the data is in little-endian byte ordering (little-endian byte ordering is where the most significant byte of data is stored at the most significant address) and converts the data to whatever byte ordering is native on this controller's architecture. Or it takes the data that is byte ordered for this controller's architecture and converts the data to little-endian byte ordering. (Notice that these two conversions are identical.)

Note Depending on the native byte ordering of the controller in use (either little-endian, or big-endian), that either the `ibeswap` or `ileswap` functions will always be a no-op, and the other will always swap bytes, as appropriate.

In all three functions, the *addr* parameter specifies a pointer to the data. The *length* parameter provides the length of the data in bytes. The *datasize* must be one of the values 1, 2, 4, or 8. It specifies the size of the data in bytes and the size of the byte swapping to perform:

- 1 = byte data and no swapping is performed.
- 2 = 16-bit word data and bytes are swapped on word boundaries.
- 4 = 32-bit longword data and bytes are swapped on longword boundaries.
- 8 = 64-bit data and bytes are swapped on 8-byte boundaries.

The *length* parameter must be an integer multiple of *datasize*. If not, unexpected results will occur.

IEEE 488.2 specifies the default data transfer format to transfer data in big-endian format. Non-488.2 devices may send data in either big-endian or little-endian format.

Note These functions do not depend on a SICL session *id*. Therefore, they may be used to perform non-SICL related task (namely, file I/O).

The following constants are available for use by your application to determine which byte ordering is native to this controller's architecture.

`I_ORDER_LE`	This constant is defined if the native controller is little-endian.
`I_ORDER_BE`	This constant is defined if the native controller is big-endian.

These constants may be used in `#if` or `#ifdef` statements to determine the byte ordering requirements of this controller's architecture. This information can then be used with the known byte ordering of the devices being used to determine the swapping that needs to be performed.

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

See Also "IPOKE", "IPEEK", "ISCANF", "IPRINTF"

ITERMCHR

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

C Syntax

```
#include <sicl.h>

int itermchr (id, tchr);
INST id;
int tchr;
```

Description By default, a successful `iread` only terminates when it reads *bufsize* number of characters, or it reads a byte with the END indicator. The `itermchr` function permits you to define a termination character condition.

The *tchr* argument is the character specifying the termination character. If *tchr* is between 0 and 255, then `iread` terminates when it reads the specified character. If *tchr* is -1, then no termination character exists, and any previous termination character is removed.

Calling `itermchr` affects all further calls to `iread` and `ifread` until you make another call to `itermchr`. The default termination character is -1, meaning no termination character is defined.

Note The `iscanf` function terminates reading on an END indicator or the termination chqaracter specified by `itermchr`.

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

See Also "IREAD", "IFREAD", "IGETTERMCHR"

ITIMEOUT

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

C Syntax

```
#include <sicl.h>

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

Description The `itimeout` function is used to set the maximum time to wait for an I/O operation to complete. In this function, *tval* defines the timeout in milliseconds. A value of zero (0) disables timeouts.

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

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

See Also "IGETTIMEOUT"

ITRIGGER

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

C Syntax

```
#include <sicl.h>

int itrigger (id);
INST id;
```

Description The `itrigger` function is used to send a trigger to a device.

Triggers on GPIB

GPIB Device Session Triggers. The `itrigger` function performs an addressed GPIB group execute trigger (GET).

GPIB Interface Session Triggers. The `itrigger` function performs an unaddressed GPIB group execute trigger (GET). The `itrigger` command on a GPIB interface session should be used in conjunction with `igpibsendcmd`.

Triggers on GPIO

GPIO Interface Session Triggers. The `itrigger` function performs the same function as calling `ixtrig` with the `I_TRIG_STD` value passed to it: it pulses the CTL0 control line.

Triggers on RS-232 (Serial)

RS-232 Device Session Triggers. The `itrigger` function sends the 488.2 `*TRG\n` command to the serial device.

RS-232 Interface Session Triggers. The `itrigger` function performs the same function as calling `ixtrig` with the `I_TRIG_STD` value passed to it: it pulses the DTR modem control line.

VXI Triggers **VXI Device Session Triggers.** The `itrigger` function sends a word-serial trigger command to the specified device.

Note The `itrigger` function is only supported on message-based device sessions with VXI.

VXI Interface Session Triggers. The `itrigger` function performs the same function as calling `ixtrig` with the `I_TRIG_STD` value passed to it: it causes one or more VXI trigger lines to fire. Which trigger lines are fired is determined by the `ivxitrigroute` function.

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

See Also "IXTRIG", and the interface-specific chapter in this manual for more information on trigger actions.

IUNLOCK

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

C Syntax

```
#include <sicl.h>

int iunlock (id);
INST id;
```

Description The `iunlock` function unlocks a device or interface that has been previously locked. If you attempt to perform an operation on a device or interface that is locked by another session, the call will hang until the device or interface is unlocked.

Calls to `ilock`/`iunlock` may be nested, meaning that there must be an equal number of unlocks for each lock. This means that simply calling the `iunlock` function may not actually unlock a device or interface again. For example, note how the following C code locks and unlocks devices:

```
ilock(id);      /* Device locked */
iunlock(id);    /* Device unlocked */

ilock(id);      /* Device locked */
   ilock(id);     /* Device locked */
   iunlock(id);   /* Device still locked */
iunlock(id);    /* Device unlocked */
```

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

See Also "ILOCK", "ISETLOCKWAIT", "IGETLOCKWAIT"

IUNMAP

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

Note Not recommended for new program development. Use IUNMAPX instead.

C Syntax

```
#include <sicl.h>

int iunmap (id, addr, map_space, pagestart, pagecnt);
INST id;
char *addr;
int map_space;
unsigned int pagestart;
unsigned int pagecnt;
```

Note Not supported over LAN.

Description The `iunmap` function unmaps a mapped memory space. The *id* specifies a VXI interface or device session. The *addr* argument contains the address value returned from the `imap` call. The *pagestart* argument indicates the page within the given memory space where the memory mapping starts. The *pagecnt* argument indicates how many pages to free.

The *map_space* argument contains the following legal values:

`I_MAP_A16`	Map in VXI A16 address space.
`I_MAP_A24`	Map in VXI A24 address space.
`I_MAP_A32`	Map in VXI A32 address space.
`I_MAP_VXIDEV`	Map in VXI device registers. (Device session only.)
`I_MAP_EXTEND`	Map in VXI A16 address space. (Device session only.)
`I_MAP_SHARED`	Map in VXI A24/A32 memory that is physically located on this device (sometimes called local shared memory).

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

See Also "IMAP"

IUNMAPX

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

C Syntax

```
#include <sicl.h>

int iunmapx (id, handle, mapspace, pagestart, pagecnt);
   INST id;
   unsigned long handle;
   int mapspace;
   unsigned int pagestart;
   unsigned int pagecnt;
```

Note Not supported over LAN.

Description The `iunmapx` function unmaps a mapped memory space. The *id* specifies a VXI interface or device session. The *addr* argument contains the address value returned from the `imap` call. The *pagestart* argument indicates the page within the given memory space where the memory mapping starts. The *pagecnt* argument indicates how many pages to free. The *map_space* argument contains the following legal values:

`I_MAP_A16`	Map in VXI A16 address space.
`I_MAP_A24`	Map in VXI A24 address space.
`I_MAP_A32`	Map in VXI A32 address space.
`I_MAP_VXIDEV`	Map in VXI device registers. (Device session only.)
`I_MAP_EXTEND`	Map in VXI A16 address space. (Device session only.)
`I_MAP_SHARED`	Map in VXI A24/A32 memory that is physically located on this device (called local shared memory).

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

See Also "IMAPX"

IVERSION

C Syntax

```
#include <sicl.h>

int iversion (siclversion, implversion);
int *siclversion;
int *implversion;
```

Description The `iversion` function stores in *siclversion* the current SICL revision number times ten that the application is currently linked with. The SICL version number is a constant defined in `sicl.h` for C. This function stores in *implversion* an implementation specific revision number (the version number of this implementation of the SICL library).

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

IVXIBUSSTATUS

Supported sessions: . . . . . interface

C Syntax

```
#include <sicl.h>

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

Description The `ivxibusstatus` function returns the status of the VXI interface. This function takes one of the following parameters in the request parameter and returns the status in the *result* parameter.

`I_VXI_BUS_TRIGGER`	Returns a bit-mask corresponding to the trigger lines which are currently being driven active by a device on the VXI bus.
`I_VXI_BUS_LADDR`	Returns the logical address of the VXI interface (viewed as a device on the VXI bus).
`I_VXI_BUS_SERVANT_AREA`	Returns the servant area size of this device.
`I_VXI_BUS_NORMOP`	Returns 1 if in normal operation, and a 0 otherwise.
`I_VXI_BUS_CMDR_LADDR`	Returns logical address of this device's commander, or -1 if no commander is present (either this device is the top level commander, or normal operation has not been established.
`I_VXI_BUS_MAN_ID`	Returns the manufacturer's ID of this device.
`I_VXI_BUS_MODEL_ID`	Returns the model ID of this device.

`I_VXI_BUS_PROTOCOL`	Returns the value stored in this device's protocol register.
`I_VXI_BUS_XPROT`	Returns the value that this device will use to respond to a *read protocol* word-serial command.
`I_VXI_BUS_SHM_SIZE`	Returns the size of VXI memory available on this device. For A24 memory, this value represents 256 byte pages. For A32 memory, this value represents 64 Kbyte pages. Interpret as an unsigned integer for this command.
`I_VXI_BUS_SHM_ADDR_SPACE`	Returns either `24` or `32` depending on whether the device's VXI memory is located in A24 or A32 memory space.
`I_VXI_BUS_SHM_PAGE`	Returns the location of the device's VXI memory. For A24 memory, the *result* is in 256 byte pages. For A32 memory, the *result* is in 64 Kbyte pages.

`I_VXI_BUS_VXIMXI`	Returns 0 if device is a VXI device. Returns 1 if device is a MXI device.
`I_VXI_BUS_TRIGSUPP`	Returns a numeric value indicating which triggers are supported. The numeric value is the sum of the following values:

I_TRIG_STD	0x00000001L
I_TRIG_ALL	0xffffffffL
I_TRIG_TTL0	0x00001000L
I_TRIG_TTL1	0x00002000L
I_TRIG_TTL2	0x00004000L
I_TRIG_TTL3	0x00008000L
I_TRIG_TTL4	0x00010000L
I_TRIG_TTL5	0x00020000L
I_TRIG_TTL6	0x00040000L
I_TRIG_TTL7	0x00080000L
I_TRIG_ECL0	0x00100000L
I_TRIG_ECL1	0x00200000L
I_TRIG_ECL2	0x00400000L
I_TRIG_ECL3	0x00800000L
I_TRIG_EXT0	0x01000000L
I_TRIG_EXT1	0x00200000L
I_TRIG_EXT2	0x00400000L
I_TRIG_EXT3	0x00800000L
I_TRIG_CLK0	0x10000000L
I_TRIG_CLK1	0x20000000L
I_TRIG_CLK2	0x40000000L
I_TRIG_CLK10	0x80000000L
I_TRIG_CLK100	0x00000800L
I_TRIG_SERIAL_DTR	0x00000400L
I_TRIG_SERIAL_RTS	0x00000200L
I_TRIG_GPIO_CTL0	0x00000100L
I_TRIG_GPIO_CTL1	0x00000080L

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

See Also "IVXITRIGON", "IVXITRIGOFF"

IVXIGETTRIGROUTE

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

C Syntax

```
#include <sicl.h>

int ivxigettrigroute (id, which, route);
INST id;
unsigned long which;
unsigned long *route;
```

Description The `ivxigettrigroute` function returns in *route* the current routing of the *which* parameter. See the `ivxitrigroute` function for more details on routing and the meaning of *route*.

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

See Also "IVXITRIGON", "IVXITRIGOFF", "IVXITRIGROUTE", "IXTRIG"

IVXIRMINFO

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

C Syntax

```
#include <sicl.h>

int ivxirminfo (id, laddr, info);
INST id;
int laddr;
struct vxiinfo *info;
```

Description The `ivxirminfo` function returns information about a VXI device from the VXI Resource Manager. The *id* is the `INST` for any open VXI session. The *laddr* parameter contains the logical address of the VXI device. The *info* parameter points to a structure of type `struct vxiinfo`. The function fills in the structure with the relevant data.

The structure `struct vxiinfo` (defined in the file `sicl.h`) is listed on the following pages.

For C programs, the `vxiinfo` structure has the following syntax:

```
struct vxiinfo {
   /* Device Identification */
   short laddr;              /* Logical Address */
   char name[16];            /* Symbolic Name (primary) */
   char manuf_name[16];      /* Manufacturer Name */
   char model_name[16];      /* Model Name */
   unsigned short man_id;    /* Manufacturer ID */
   unsigned short model;     /* Model Number */
   unsigned short devclass;  /* Device Class */

   /* Self Test Status */
   short selftest;           /* 1=PASSED 0=FAILED */

   /* Location of Device */
   short cage_num;           /* Card Cage Number */
   short slot;       /* Slot #, -1 is unknown, -2 is MXI */
   /* Device Information */
   unsigned short protocol;  /* Value of protocol register
*/
   unsigned short x_protocol;/* Value from Read Protocol
```

```
command */
   unsigned short servant_area;/* Value of servant area */

   /* Memory Information */
   /* page size is 256 bytes for A24 and 64K bytes for
A32*/
   unsigned short addrspace;/* 24=A24, 32=A32, 0=none */
   unsigned short memsize;/* Amount of memory in pages */
   unsigned short memstart;/* Start of memory in pages */

   /* Misc. Information */
   short slot0_laddr;  /* LU of slot 0 device, -1 if unknown
*/
   short cmdr_laddr;  /* LU of commander, -1 if top level*/

   /* Interrupt Information */
   short int_handler[8];    /* List of interrupt handlers */
   short interrupter[8];    /* List of interrupters */
   short file[10];          /* Unused */
}
```

This static data is set up by the VXI resource manager.

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

See Also See the platform-specific manual for the section on the Resource Manager.

IVXISERVANTS

Supported sessions: . . . . . interface

C Syntax

```
#include <sicl.h>

int ivxiservants (id, maxnum, list);
INST id;
int maxnum;
int *list;
```

Description The `ivxiservants` function returns a list of VXI servants. This function returns the first *maxnum* servants of this controller. The *list* parameter points to an array of integers that holds at least *maxnum* integers. This function fills in the array from beginning to end with the list of active VXI servants. All unneeded elements of the array are filled with -1.

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

IVXITRIGOFF

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

C Syntax

```
#include <sicl.h>

int ivxitrigoff (id, which);
INST id;
unsigned long which;
```

Description The `ivxitrigoff` function de-asserts trigger lines and leaves them deactivated. The *which* parameter uses all of the same values as the `ixtrig` command, namely:

`I_TRIG_ALL`	All standard triggers for this interface (that is, the bitwise OR of all valid triggers)
`I_TRIG_TTL0`	TTL Trigger Line 0
`I_TRIG_TTL1`	TTL Trigger Line 1
`I_TRIG_TTL2`	TTL Trigger Line 2
`I_TRIG_TTL3`	TTL Trigger Line 3
`I_TRIG_TTL4`	TTL Trigger Line 4
`I_TRIG_TTL5`	TTL Trigger Line 5
`I_TRIG_TTL6`	TTL Trigger Line 6
`I_TRIG_TTL7`	TTL Trigger Line 7
`I_TRIG_ECL0`	ECL Trigger Line 0
`I_TRIG_ECL1`	ECL Trigger Line 1
`I_TRIG_ECL2`	ECL Trigger Line 2
`I_TRIG_ECL3`	ECL Trigger Line 3

`I_TRIG_EXT0`	External BNC or SMB Trigger Connector 0
`I_TRIG_EXT1`	External BNC or SMB Trigger Connector 1

Any combination of values may be used in *which* by performing a bit-wise OR of the desired values.

Note To simply fire trigger lines (assert then de-assert the lines), use `ixtrig` instead of `ivxitrigon` and `ivxitrigoff`.

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

See Also "IVXITRIGON", "IVXITRIGROUTE", "IVXIGETTRIGROUTE", "IXTRIG"

IVXITRIGON

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

C Syntax

```
#include <sicl.h>
int ivxitrigon (id, which);
INST id;
unsigned long which;
```

Description The `ivxitrigon` function asserts trigger lines and leaves them activated. The *which* parameter uses all of the same values as the `ixtrig` command, namely:

`I_TRIG_ALL`	All standard triggers for this interface (that is, the bitwise OR of all valid triggers)
`I_TRIG_TTL0`	TTL Trigger Line 0
`I_TRIG_TTL1`	TTL Trigger Line 1
`I_TRIG_TTL2`	TTL Trigger Line 2
`I_TRIG_TTL3`	TTL Trigger Line 3
`I_TRIG_TTL4`	TTL Trigger Line 4
`I_TRIG_TTL5`	TTL Trigger Line 5
`I_TRIG_TTL6`	TTL Trigger Line 6
`I_TRIG_TTL7`	TTL Trigger Line 7
`I_TRIG_ECL0`	ECL Trigger Line 0
`I_TRIG_ECL1`	ECL Trigger Line 1
`I_TRIG_ECL2`	ECL Trigger Line 2
`I_TRIG_ECL3`	ECL Trigger Line 3
`I_TRIG_EXT0`	External BNC or SMB Trigger Connector 0
`I_TRIG_EXT1`	External BNC or SMB Trigger Connector 1

Any combination of values may be used in *which* by performing a bit-wise OR of the desired values.

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

See Also "IVXITRIGOFF", "IVXITRIGROUTE", "IVXIGETTRIGROUTE", "IXTRIG"

IVXITRIGROUTE

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

C Syntax

```
#include <sicl.h>

int ivxitrigroute (id, in_which, out_which);
INST id;
unsigned long in_which;
unsigned long out_which;
```

Description The `ivxitrigroute` function routes VXI trigger lines. With some VXI interfaces, it is possible to route one trigger input to several trigger outputs.

The *in_which* parameter may contain only one of the valid trigger values. The *out_which* may contain zero, one, or several of the following valid trigger values:

`I_TRIG_ALL`	All standard triggers for this interface (that is, the bit-wise OR of all valid triggers) (*out_which* ONLY)
`I_TRIG_TTL0`	TTL Trigger Line 0
`I_TRIG_TTL1`	TTL Trigger Line 1
`I_TRIG_TTL2`	TTL Trigger Line 2
`I_TRIG_TTL3`	TTL Trigger Line 3
`I_TRIG_TTL4`	TTL Trigger Line 4
`I_TRIG_TTL5`	TTL Trigger Line 5
`I_TRIG_TTL6`	TTL Trigger Line 6
`I_TRIG_TTL7`	TTL Trigger Line 7
`I_TRIG_ECL0`	ECL Trigger Line 0
`I_TRIG_ECL1`	ECL Trigger Line 1

`I_TRIG_ECL2`	ECL Trigger Line 2
`I_TRIG_ECL3`	ECL Trigger Line 3
`I_TRIG_EXT0`	External BNC or SMB Trigger Connector 0
`I_TRIG_EXT1`	External BNC or SMB Trigger Connector 1

The *in_which* parameter may also contain:

`I_TRIG_CLK0`	Internal clocks provided by the controller (implementation- specific)
`I_TRIG_CLK1`	Internal clocks provided by the controller (implementation- specific)
`I_TRIG_CLK2`	Internal clocks provided by the controller (implementation- specific)

This function routes the trigger line in the *in_which* parameter to the trigger lines contained in the *out_which* parameter. In other words, when the line contained in *in_which* fires, all of the lines contained in *out_which* are also fired.

For example, the following command causes EXT0 to fire whenever TTL3 fires:

```
ivxitrigroute(id, I_TRIG_TTL3, I_TRIG_EXT0);
```

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

See Also "IVXITRIGON", "IVXITRIGOFF", "IVXIGETTRIGROUTE", "IXTRIG"

IVXIWAITNORMOP

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

C Syntax

```
#include <sicl.h>

int ivxiwaitnormop (id);
INST id;
```

Description The `ivxiwaitnormop` function is used to suspend the process until the interface or device is active (that is, establishes normal operation). See the `iwaithdlr` function for other methods of waiting for an interface to become ready to operate.

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

See Also "IWAITHDLR", "IONINTR", "ISETINTR", "ICLEAR"

IVXIWS

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

C Syntax

```
#include <sicl.h>

int ivxiws(id,wscmd,wsresp,rpe);
INST id;
unsigned short wscmd;
unsigned short *wsresp;
unsigned short *rpe;
```

Description The `ivxiws` function sends a word-serial command to a VXI message-based device. The *wscmd* contains the word-serial command. If *wsresp* contains zero (0), then this function does not read a word-serial response. If *wsresp* is non-zero, then the function reads a word-serial response and stores it in that location. If `ivxiws` executes successfully, *rpe* does not contain valid data. If the word-serial command errors, *rpe* contains the Read Protocol Error response, the `ivxiws` function returns `I_ERR_IO`, and the *wsresp* parameter contains invalid data.

Note The `ivxiws` function will always try to read the response data if the *wsresp* parameter is non-zero. If you send a word serial command that does not return response data, and the *wsresp* argument is non-zero, this function will hang or timeout (see `itimeout`) waiting for the response.

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

See Also "ITIMEOUT"

IWAITHDLR

C Syntax

```
#include <sicl.h>

int iwaithdlr (timeout);
long timeout;
```

Description The `iwaithdlr` function causes the process to suspend until either an enabled SRQ or interrupt condition occurs and the related handler executes. Once the handler completes its operation, this function returns and processing continues.

If *timeout* is non-zero, then iwaithdlr terminates and generates an error if no handler executes before the given time expires. If *timeout* is zero, then `iwaithdlr` waits indefinitely for the handler to execute.

Specify *timeout* in milliseconds.

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

The `iwaithdlr` function will implicitly enable interrupts. In other words, if you have called `iintroff`, `iwaithdlr` will re-enable interrupts, then disable them again before returning.

Note Interrupts should be disabled if you are using `iwaithdlr`. Use `iintroff` to disable interrupts.

The reason for disabling interrupts is because there is a race condition between the `isetintr` and `iwaithdlr`, and, if you only expect one interrupt, it might come before the `iwaithdlr` executes.

The interrupts will still be disabled after the `iwaithdlr` function has completed.

For example:

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

In a multi-threaded application, `iwaithdlr` will enable interrupts for the whole process. If two threads call `iintroff`, and one of them then calls `iwaithdlr`, interrupts will be enabled and both threads can receive interrupt events. Note that this is not a defect, since your application must handle the enabling/disabling of interrupts and keep track of when all threads are ready to receive interrupts.

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

See Also "IONINTR", "IGETONINTR", "IONSRQ", "IGETONSRQ", "IINTROFF", "IINTRON"

IWRITE

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

C Syntax

```
#include <sicl.h>

int iwrite (id, buf, datalen, endi, actualcnt);
INST id;
char *buf;
unsigned long datalen;
int endi;
unsigned long *actualcnt;
```

Description

The `iwrite` function is used to send a block of data to an interface or device. This function writes the data specified in *buf* to the session specified in *id*. The *buf* argument is a pointer to the data to send to the specified interface or device. The *datalen* argument is an unsigned long integer containing the length of the data block in bytes.

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

The *actualcnt* argument is a pointer to an unsigned long integer which, upon exit, will contain the actual number of bytes written to the specified interface or device. A NULL pointer can be passed for this argument and no value will be written.

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

Return Value

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

See Also

"IREAD", "IFREAD", "IFWRITE"

IXTRIG

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

C Syntax

```
#include <sicl.h>

int ixtrig (id, which);
INST id;
unsigned long which;
```

Description The ixtrig function is used to send an extended trigger to an interface. The argument which can be:

`I_TRIG_STD`	Standard trigger operation for all interfaces. The exact operation of I_TRIG_STD depends on the particular interface. See the following subsections for interface-specific information.
`I_TRIG_ALL`	All standard triggers for this interface (that is, the bit-wise OR of all supported triggers).
`I_TRIG_TTL0`	TTL Trigger Line 0
`I_TRIG_TTL1`	TTL Trigger Line 1
`I_TRIG_TTL2`	TTL Trigger Line 2
`I_TRIG_TTL3`	TTL Trigger Line 3
`I_TRIG_TTL4`	TTL Trigger Line 4
`I_TRIG_TTL5`	TTL Trigger Line 5
`I_TRIG_TTL6`	TTL Trigger Line 6
`I_TRIG_TTL7`	TTL Trigger Line 7
`I_TRIG_ECL0`	ECL Trigger Line 0
`I_TRIG_ECL1`	ECL Trigger Line 1

`I_TRIG_ECL2`	ECL Trigger Line 2
`I_TRIG_ECL3`	ECL Trigger Line 3
`I_TRIG_EXT0`	External BNC or SMB Trigger Connector 0
`I_TRIG_EXT1`	External BNC or SMB Trigger Connector 1
`I_TRIG_EXT2`	External BNC or SMB Trigger Connector 2
`I_TRIG_EXT3`	External BNC or SMB Trigger Connector 3

Triggers on GPIB

When used on a GPIB interface session, passing the `I_TRIG_STD` value to the `ixtrig` function causes an unaddressed GPIB group execute trigger (GET). The `ixtrig` command on a GPIB interface session should be used in conjunction with the `igpibsendcmd`. There are no other valid values for the `ixtrig` function.

Triggers on GPIO

The `ixtrig` function will pulse either the CTL0 or CTL1 control line. The following values can be used:

`I_TRIG_STD`	CTL0
`I_TRIG_GPIO_CTL0`	CTL0
`I_TRIG_GPIO_CTL1`	CTL1

Triggers on RS-232 (Serial)

The `ixtrig` function will pulse either the DTR or RTS modem control lines. The following values can be used:

`I_TRIG_STD`	Data Terminal Ready (DTR)
`I_TRIG_SERIAL_DTR`	Data Terminal Ready (DTR)
`I_TRIG_SERIAL_RTS`	Ready To Send (RTS)

Triggers on VXI When used on a VXI interface session, passing the `I_TRIG_STD` value to the `ixtrig` function causes one or more VXI trigger lines to fire. Which trigger lines are fired is determined by the `ivxitrigroute` function. The `I_TRIG_STD` value has no default value. Therefore, if it is not defined before it is used, no action will be taken.

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

See Also "ITRIGGER", "IVXITRIGON", "IVXITRIGOFF"

_SICLCLEANUP

C Syntax

```
#include <sicl.h>

int _siclcleanup(void);
```

Description This routine is called when a program is done with all SICL I/O resources. Calling this routine is not required on HP-UX.

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

A

The SICL Files

The SICL Files

This appendix list the files and directories created on your system for SICL on HP-UX 11i and Linux.

SICL-RUN Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-RUN`		Files for customization.
`/opt/sicl`	`/opt/sicl`	The main SICL software directory.
`/opt/sicl/bin`	`/opt/sicl/bin`	The SICL configuration tools, programs, etc.
`/opt/sicl/defaults`	`/opt/sicl/defaults`	Default versions of the hwconfig.cf and iproc.cf files.
`/opt/sicl/lib`	`/opt/sicl/lib`	Driver binary modules, which are linked and inserted in the kernel by the SICL configuration programs. Also adds the shared libraries.
`/opt/sicl/lib`	`/opt/sicl/lib`	Files for the shared library.

SICL-PRG Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/SICL/SICL-PRG		Files for customization.
/opt/sicl/lib	/opt/sicl/lib	The SICL library (libsicl.a).
/opt/sicl/include	/opt/sicl/include	The SICL header file (sicl.h).
/opt/sicl	/opt/sicl	The DIL to SICL migration document.
/opt/sicl/bin	/opt/sicl/bin	The dil2sicl migration tool.
/opt/sicl/share/examples	/opt/sicl/share/examples	The SICL example programs.

SICL-VXI Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/SICL/SICL-VXI		Files for customization.
/opt/sicl/bin	/opt/sicl/bin	The VXI specific configuration files, including the resource manager program, ivxirm.
/opt/sicl/defaults	/opt/sicl/defaults	The default versions of the VXI configuration files.
/opt/sicl/lib	/opt/sicl/lib	The VXI driver binary modules, which are linked and inserted in the kernel by the SICLconf configuration program. Also adds the driver shared libraries.

SICL-HPIB Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-HPIB`		Files for customization.
`/opt/sicl/defaults`	`/opt/sicl/defaults`	The default versions of the GPIB configuration files.
`/opt/sicl/lib`		The GPIB driver binary modules, which are linked and inserted in the kernel by the SICL configuration programs. Also adds the driver shared libraries.

SICL-GPIO Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-GPIO`		Files for customization.
`/opt/sicl/defaults`	`/opt/sicl/defaults`	The default versions of the GPIO configuration files.
`/opt/sicl/lib`		The GPIO driver binary modules, which are linked and inserted in the kernel by the configuration program. Also adds the driver shared libraries.

SICL-RS232 Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/ SICL/SICL-RS232		Files for customization.
/opt/sicl/defaults	/opt/sicl/ defaults	The default versions of the RS-232 configuration files.
/opt/sicl/lib		The RS-232 driver binary modules, which are linked and inserted in the kernel by the SICL configuration programs. Also adds the driver shared libraries.

SICL-LAN Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/ SICL/SICL-LAN		Files for customization.
/opt/sicl/defaults	/opt/sicl/ defaults	The default versions of the LAN configuration files.
/opt/sicl/lib	/opt/sicl/lib	The LAN driver shared libraries.

SICL-LANSVR Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/ SICL/SICL-LANSVR		Files for customization.
/opt/sicl/bin	/opt/sicl/bin	The SICL LAN server daemon and a LAN configuration utility.

SICL-MAN Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/SICL/SICL-MAN		Files containing copyright information.
/opt/sicl/share/man/man1	/opt/sicl/share/man/man1	Files containing man pages for SICL user utilities.
/opt/sicl/share/man/man1m	/opt/sicl/share/man/man1m	Files containing man pages for SICL system administrator utilities.
/opt/sicl/share/man/man3	/opt/sicl/share/man/man3	Files containing man pages for SICL function calls.
/opt/sicl/share/man/man4	/opt/sicl/share/man/man4	Files containing man pages for SICL configuration files.

SICL-MAN-HPIB Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/SICL/SICL-MAN-HPIB		Files for customization.
/opt/sicl/share/man/man3	/opt/sicl/share/man/man3	Files containing man pages for SICL GPIB specific function calls.

SICL-MAN-GPIO Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/SICL/SICL-MAN-GPIO		Files for customization.
/opt/sicl/share/man/man3	/opt/sicl/share/man/man3	Files containing man pages for SICL GPIO specific function calls.

SICL-MAN-VXI Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/SICL/SICL-MAN-VXI		Files containing copyright information.
/opt/sicl/share/man/man1m	/opt/sicl/share/man/man1m	Files containing man pages for SICL VXI/MXI specific system administrator utilities.
/opt/sicl/share/man/man3	/opt/sicl/share/man/man3	Files containing man pages for SICL VXI/MXI specific function calls.
/opt/sicl/share/man/man4	/opt/sicl/share/man/man4	Files containing man pages for SICL VXI/MXI specific configuration files.

SICL-MAN-RS232 Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/SICL/SICL-MAN-RS232		Files containing copyright information.
/opt/sicl/share/man/man3	/opt/sicl/share/man/man3	Files containing man pages for SICL RS-232 specific function calls.

SICL-MAN-LAN Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/SICL/SICL-MAN-LAN		Files containing copyright information.
/opt/sicl/share/man/man3	/opt/sicl/share/man/man3	Files containing man pages for SICL LAN specific function calls.

SICL-MAN-LANSVR Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/SICL/SICL-MAN-LANSVR		Files containing copyright information.
/opt/sicl/share/man/man1m	/opt/sicl/share/man/man1m	LAN specific man pages.

SICL-DIAG Filesets

HP-UX 11i Directories	Linux Directories	Description of Files
/var/adm/sw/products/SICL/SICL-DIAG		Files containing copyright information.
/opt/sicl/bin	/opt/sicl/bin	A directory containing diagnostic programs and utilities.

B

Updating HP-UX 9 SICL Applications

Updating HP-UX 9 SICL Applications

This appendix describes what you need to do in order to run your SICL for HP-UX 9 applications on HP-UX 11i.

Building SICL Applications on HP-UX 11i

If you built your SICL application on HP-UX 9.x with the SICL shared library, then no changes are necessary. However, if you used the SICL archive library, then you must either re-build your application or run the provided script. See the following:

- If your SICL 9.x application was linked with the SICL shared library, then no modification or re-compiles are necessary.
- If your SICL 9.x application was linked with the SICL archive library, then you can do one of the following:
  - Re-compile your SICL 9.x application on HP-UX 11i with the SICL shared library. This is the recommended method. See "Compiling and Linking an SICL Program" in Chapter 2, "Getting Started with SICL," of this manual.

OR

- Execute the /opt/sicl/bin/sicl_tl script as super user to install transition links to allow you SICL 9.x executables to run without modification or re-compiling:

To create symbolic links:

```
/opt/sicl/bin/sicl_tl install
```

To remove symbolic links:

```
/opt/sicl/bin/sicl_tl remove
```

Linking with the Archive Library on HP-UX 9

Note For future compatibility, we recommend that you link with the SICL shared library as shown in Chapter 2, "Getting Started with SICL."

SICL for HP-UX 9 is shipped with both a shared library and an archive library. By default, SICL programs are built with the shared library unless you specify the archive library. The following command creates the `idn` executable file while linking in the archive library:

```
cc  -o idn idn.c -Wl,-E,-a,archive -lsicl -Wl,-a,shared -ldld
```

or

```
cc -o idn idn.c  /usr/lib/libsicl.a -Wl,-E  -ldld
```

- The `-Wl` option specifies the compile options to pass to the linker.
- The `-E` option is a linker option that exports symbols to shared libraries.
- The `-a` option is a linker option that tells the linker which type of library to use (in this case `archive`).
- The `-ldld` option links in the `dld` library for use by SICL.

C

The SICL Utilities

The SICL Utilities

This appendix describes the utilities that are shipped with SICL. The following utilities are described in alphabetical order:

- `iclear`
- `ipeek`
- `ipoke`
- `iread`
- `iwrite`

iclear

Syntax **iclear [-t** *timeout***] [-v] [-?]** *sym_name***>**

Description `iclear` performs a device or interface defined clear operation on the device or interface specified by the *sym_name* parameter. *Sym_name* is the SICL address of the device or interface being addressed. If *sym_name* refers to a device, then a device clear command will be sent to the device. If *sym_name* refers to an interface, then the interface clear command will be sent to that interface. The actual functions of the device clear or interface clear are specific to the device or interface.

For example, executing `iclear` on an GPIB device will result in the DCL command being sent to that device. Executing `iclear` on an GPIB interface will result in the IFC and REN line being pulsed (if the interface is system controller), and the interface hardware being reset.

When used on a GPIO interface session, `iclear` pulses the P_RESET line for approximately 12 microseconds, aborts any pending writes, discards any data in the receive buffer, and resets any error conditions. Optionally, it also clears the Data Out port, depending on the *mode* configuration specified during the GPIO interface configuration.

The `iclear` command, when used on a VXI/MXI interface session causes a pulse on the SYSRESET line which cancels the normal operation state until the resource manager has reconfigured the VXI system. The `iclear` command, when used on a VXI message-based device session sends a word-serial `device clear` command to the specified device.

Note If a SYSRESET (`iclear`) occurs and the `iscpi` instrument is running, then the iscpi instrument will be terminated. If this happens, you will get a `No Connect` error message and you need to re-open the iscpi communications session.

Using the `iclear` command on the RS-232 interface session clears the input and output buffers and sends a break character.

The parameter definitions follow.

`t` *timeout*	Times out after timeout milliseconds.
`v`	Turns on verbose mode.
`?`	Prints the usage of the iclear program.

Example

```
iclear -t 1000 vxi
```

ipeek

Syntax **ipeek [-v] [-?]** [-b / -w / -l] *sym_name map_space offset*

Description `ipeek` is the SICL utility for examining memory locations on interfaces that support mapping. The `ipeek` utility will print the contents of the specified memory location in hexadecimal.

The *sym_name* is the SICL `symbolic name` of the interface. The interface must support mapping, such as VXI.

The *map_space* is the map area that you would like to examine. Currently the only interfaces supported are VXI and MXI. The valid map spaces are A16, A24, A32, VXIDEV, and EXTEND. See the imap function in Chapter 10 for a description of these mappings.

The *offset* is the offset, in bytes, from the beginning of the mapped space to the location that is to be examined.

The parameter definitions follow.

`v`	Turns on verbose mode.
`?`	Prints the usage of the ipeek program.
`b`	Specifies that the register size is a byte (8 bits).
`w`	Specifies that the register size is a word (16 bits, default).
`l`	Specifies that the register size is a long (32 bits).

Example

```
ipeek vxi A16 0xC000 1
```

ipoke

Syntax **ipoke [-v] [-?]** [-b / -w / -l] *sym_name map_space offset value*

Description `ipoke` is the SICL utility for writing to memory locations on interfaces that support mapping. The `ipoke` utility will write the contents of the value parameter to the specified memory location.

The *sym_name* is the SICL `symbolic name` of the interface. The interface must support mapping, such as VXI.

The *map_space* is the map area that you would like to write to. Currently the only interfaces supported are VXI and MXI. The valid map spaces are A16, A24, A32, VXIDEV, and EXTEND. See the imap function in Chapter 10 for a description of these mappings.

The *offset* is the offset, in bytes, from the beginning of the mapped space to the location that is to be written.

The parameter definitions follow.

`v`	Turns on verbose mode.
`?`	Prints the usage of the ipoke program.
`b`	Specifies that the register size is a byte (8 bits).
`w`	Specifies that the register size is a word (16 bits, default).
`l`	Specifies that the register size is a long (32 bits).

Example

```
ipoke vxi A24 0x200000 1 0x0000
```

iread

Syntax **iread [-t** *timeout***] [-c** *count***] [-e** *end_char***] [-v] [-?]** *sym_name*

Description `iread` is the SICL utility for reading data from devices. The output of `iread` goes to stdout. The read is terminated only when *count* number of bytes is read, a timeout occurs, a byte is read with the END indicator, or the termination character *end_char* is read. These conditions may occur in combination.

The *sym_name* is the SICL `symbolic name`, or address, of the device that was determined during the interface configuration. Note that `iread` is only supported for device addresses.

The parameter definitions follow.

`t`	*timeout*	Specifies the timeout value in milliseconds.
`c`	*count*	Specifies the number of bytes to read.
`e`	*end_char*	Defines a termination character for the read.
`v`		Turns on verbose mode.
`?`		Prints the usage of the iread program.

Example

```
iread hpib,16
```

iwrite

Syntax **iwrite [-s** *size***] [-t** *timeout***] [-e 0|1] [-v] [-?]** *sym_name*

Description `iwrite` is the SICL utility for writing data to a device. The input of iwrite comes from stdin. The write is terminated only when *size* number of bytes is written or a timeout occurs.

The *sym_name* is the SICL `symbolic name` of the device. Note that `iwrite` is only supported for device addresses.

The parameter definitions follow:

`s` *size*	Specifies the number of bytes to read.
`t` *timeout*	Specifies the timeout value in milliseconds .
`e 0|1`	Set to non-zero if the END indicator should be given on the last byte of the block, or zero if it should not. Note that if this parameter is not specified, iwrite will default to giving the END indicator on the last byte of the block.
`v`	Turns on verbose mode.
`?`	Prints the usage of the iwrite program.

Example

```
iwrite hpib,16
```

D

Customizing your VXI/MXI System

Customizing your VXI/MXI

This appendix describes what files you would edit to customize your VXI/MXI system. Additionally, the VXI/MXI specific utilities are described. This chapter contains the following sections:

- Overview of VXI/MXI Configuration
- The VXI/MXI Resource Manager
- The VXI/MXI Configuration Files
- The VXI/MXI Configuration Utilities

Overview of VXI/MXI Configuration

When SICL is installed and configured according to the procedures in the *I/O Libraries Installation and Configuration Guide*, certain SICL utilities and configuration files are copied onto your system. The VXI/MXI system is configured using two SICL utilities and the VXI/MXI configuration files. These utilities automatically run when the system boots. The following is a summary of the VXIbus boot process utilities:

`iproc`	This utility runs at system boot and performs various system initialization functions. It uses the `iproc.cf` configuration file to determine when the other configuration utility, `ivxirm`, runs.
`ivxirm`	This utility runs the resource manager which initializes and configures the VXI/MXI cardcage resources. The resource manager reads the VXI/MXI configuration files and polls the VXI devices to determine their resources and capabilities. This utility runs at cardcage initialization unless otherwise specified in the `iproc.cf` configuration file (default is to run at cardcage initialization).
configuration files	These files specify some site-dependent configuration rules and any changes from the default.

Note These utilities and configuration files are only provided with the `SICL-VXI` fileset on HP-UX 11i. In order to use VXI/MXI, you must have loaded this fileset during the installation. See the *I/O Libraries Installation and Configuration Guide* for more details. The utilities and configuration files are described in more detail in the sections that follow.

The VXI/MXI Resource Manager (ivxirm)

The `ivxirm` utility is the resource manager which initializes and configures the VXI/MXI cardcage resources. The resource manager reads the VXI/MXI configuration files and polls the VXI devices to determine their resources and capabilities. The commander servant hierarchy is set up and the appropriate commands are sent to the VXI devices. The information is then stored in the following file:

`/etc/opt/sicl/vxi`*LU*`/rsrcmgr.out`

Where *LU* is the logical unit of the VXI/MXI interface. The resource manager also optionally prints this information to the standard output.

You can run this utility from the command line, or it generally runs at cardcage initialization if specified in the `iproc.cf` configuration file (default is to run when the system boots).

Additionally, there is another utility that can be used to review the system resources. The `ivxisc` utility reads the `rsrcmgr.out` file and prints a human readable display of the current configuration. See the `ivxirm` and `ivxisc` utilities later in this chapter for a description on using these utilities.

The VXI/MXI Configuration Files

In general, the resource manager follows a set of rules defined by the VXI Standard when configuring the system. However, the VXI standard does not define some aspects of configuration and sometimes you need to make changes to the default.

The VXI/MXI configuration files specify some site-dependent configuration rules and any changes from the default. These files reside in the following directories:

VXI/MXI Configuration Files

File Name	Directory Location
`vximanuf.cf`	`/opt/sicl`
`vximodel.cf`	`/opt/sicl`
`dynamic.cf`	`/etc/opt/sicl/vxi`*LU*
`vmedev.cf`	`/etc/opt/sicl/vxi`*LU*
`irq.cf`	`/etc/opt/sicl/vxi`*LU*
`cmdrsrvt.cf`	`/etc/opt/sicl/vxi`*LU*
`names.cf`	`/etc/opt/sicl/vxi`*LU*
`oride.cf`	`/etc/opt/sicl/vxi`*LU*
`ttltrig.cf`	`/etc/opt/sicl/vxi`*LU*

Where *LU* is the logical unit of the VXI/MXI interface. Each file is explained in the following sections.

The vximanuf.cf Configuration File

The `vximanuf.cf` file contains a database that cross references the VXI manufacturer id numbers and the name of the manufacturer. The `ivxirm` utility reads the manufacturer id number from the VXI device. The ivxisc utility then uses that number and this file to print out the name of the manufacturer. If you add a new VXI vendor that is not currently in the file, you may want to add an entry to the file.

The vximodel.cf Configuration File

The `vximodel.cf` file contains a database that lists a cross reference of manufacturer id, model id, and VXI device names. The `ivxirm` utility reads the model id number from the VXI device and the `ivxisc` utility uses that information and this file to print out the VXI device model. If you add a new VXI device to your system that is not currently in this database, you may want to add an entry to this file.

The dynamic.cf Configuration File

The `dynamic.cf` file contains a list of VXI devices to be dynamically configured. You only need to add entries to this file if you want to override the default dynamic configuration assignment by the resource manager. Normally, if you have a dynamically configurable device and the logical address is set at 255, the resource manager will assign the first available address. However, if a dynamically configurable device has an entry in this file, the resource manager will assign the address listed in the file.

The vmedev.cf Configuration File

The `vmedev.cf` file contains a list of VME devices that use resources in the VXI cardcage. Since the resource manager is unable to detect VME devices, the resource manager uses this information to determine such things as the slot number, where the VME device is located (A16, A32, or A24), how much memory it uses, and what interrupt lines it uses. Additionally, the resource manager verifies that multiple resources aren't allocated. See "Communicating with VME Devices" in chapter 6, "Using SICL with VXI/MXI," for more information on setting up VME devices in your VXI cardcage. This file is also used by the `ivxisc` utility to print out information about the devices.

The irq.cf Configuration File

The `irq.cf` file is a database that maps specific interrupt lines to VXI interrupt handlers. If you have non-programmable interrupters and you want the interrupters to be recognized by a VXI interrupt handler, you must make an entry in this file. Additionally, if you have programmable interrupters and you want them to be recognized by a device other than what's assigned by the resource manager (the commander of that device), you can make an entry in this file to override the default. Keep in mind that not all VXI devices need to use interrupt lines and not all interrupt lines need to be assigned. Note that any interrupt lines assigned in this file cannot also be assigned in the `vmedev.cf` configuration file.

The cmdrsrvt.cf Configuration File

The `cmdrsrvt.cf` file contains a commander/servant hierarchy other than the default for the VXI system. The resource manager will set up the commander/servant hierarchy according to the commander's logical addresses and the servant area switch. However, you can use this file to override the default according to the commander's switch settings. This file should only contain changes from the normal.

The names.cf Configuration File

The `names.cf` file is a database that contains a list of symbolic names to assign VXI devices that have been configured. The `ivxirm` utility reads the model id number from the VXI device and the `ivxisc` utility uses that information and this file to print out the VXI device symbolic name. If you add a new VXI device to your system that is not currently in the database, you may want to add an entry to this file.

The oride.cf Configuration File

The `oride.cf` file contains values to be written to logical address space for register-based instruments. This data is written to A16 address space after the resource manager runs, but before the system's resources are released. This can be used for custom configuration of register-based instruments every time the resource manager runs. It can also be used to program extender devices like the VXI/MXI Bus Extender card. See "Routing External Trigger Lines on the E1482 VXI-MXI Extender Bus Card" in Chapter 6, "Using SICL with VXI/MXI," for an example of using this file.

The ttltrig.cf Configuration File

The `ttltrig.cf` file contains the mapping of VXI devices to TTL trigger lines for extended VXI/MXI systems. If you have an extended VXI/MXI system and you want your TTL trigger lines to be recognized, you must map the TTL trigger line to the source logical address in this file. This file can only be used for extended VXI systems. See "Routing VXI TTL Trigger Lines in a VXI/MXI System" in Chapter 6, "Using SICL with VXI/MXI," for an example of using this file.

The iproc Utility (Initialization and SYSRESET)

On HP-UX and Linux systems, SICL installs a program called `iproc`. This program uses the `iproc.cf` file to determine how your system is initialized. The `iproc.cf` file determines when the `ivxirm` program runs and with what options. Additionally, the `iproc.cf` file specifies what action is taken when your VXI system encounters a SYSRESET.

If you have a VXI backplane, the `iproc` program is run at system boot time. This program becomes a daemon and monitors the VXI backplane for SYSRESET. The `iproc.cf` file tells `iproc` what to do if a SYSRESET occurs. Usually you want the resource manager to run and configure your system (since the SYSRESET has invalidated the configuration).

The `iproc.cf` file is stored in the following directory:

```
/etc/opt/sicl
```

Note If a SYSRESET (power down or `iclear`) occurs and the `iscpi` instrument is running, then the `iscpi` instrument server task will be killed. If this happens, you will get a `No Connect` error message and you need to re-open the `iscpi` communications session.

Note The SYSRESET line is commented out by default. You *must* un-comment the following line in the `/etc/opt/sicl/iproc.cf` file in order for the resource manager to run on SYSRESET.

```
sysreset vxi ivxirm -t 5&
```

The following is an example of the `/etc/opt/sicl/iproc.cf` file:

```
#
# iproc configuration file
#

#
# Boot up functions
#
# Lines are of the form:
#       boot <command_to_execute>
#
boot echo "SICL:  Instrument I/O Initialization"

# The next line must always exist.
boot siclsetup

#
# V743 or VXI/MXI Support
#

#boot ivxirm -p -I vxi

# When a SYSRESET occurs, rerun the resource manager
  (delay 5 sec).
# The resource manager MUST be run in the background
  (i.e. last
# character should be a '&').

#sysreset vxi ivxirm -t 5&

# Sample lines for a second VXI/MXI interface:
#boot ivxirm -p -I vxi2
#sysreset vxi2 ivxirm -I vxi2 -t 5&

# The following line must be present for ALL VXI/MXI
  systems
#monitor
```

Viewing the VXIbus System Configuration

You can use the SICL `ivxisc` utility to read the current system configuration and print a human readable display:

```
ivxisc /etc/opt/sicl/vxiLU
```

LU represents the logical unit of the VXI/MXI interface. Run the I/O setup configuration utility for information on the `Logical Unit` of your VXI/MXI interface. Also see "VXI/MXI Utilities" later in this chapter for information on using this utility.

VXI/MXI Configuration Utilities

The following SICL utilities are available to help you configure your VXI/MXI system:

- `iproc`
- `itrginvrt`
- `ivxirm`
- `ivxisc`

The utilities are located in the following directory:

```
/opt/sicl/bin
```

Each of these utilities is described in detail in the sections that follow.

iproc

Description

`iproc` is designed to run at system boot time. It performs various SICL system initialization functions including the creation of SICL device files. `/dev/sicl` contains device files. In addition, it is configurable by the system administrator to execute programs at boot time or on certain asynchronous events, such as VXI SYSRESET. This configuration is done by editing the file `iproc.cf`, which is read only when the `iproc` daemon begins execution. It consists of lines beginning with keywords which determine the actions of the `iproc` program. The `iproc.cf` file is located in the following directory:

```
/etc/opt/sicl
```

Note Only one iproc daemon is allowed to be running on a specific system.

The format of the configuration lines is as follows:

keyword action
or
keyword interface name action

The functions of the keywords are described below:

`boot` — This keyword will execute the action when the `iproc` daemon begins execution. The normal time for `iproc` to run is when the system boots.

`sysreset` *interface_name*	This keyword will execute the action on the *interface_name* when a VXI SYSRESET interrupt is detected by the `iproc` daemon. This function is primarily used to ensure that the VXI resource manager, `ivxirm`, will be run in response to a VXI SYSRESET. This requires `iproc` to continue execution.
`monitor`	This keyword allows the `iproc` daemon to continue execution when there are no other keywords, like sysreset, which would require it to continue execution.

Note Without a keyword in `iproc.cf` that allows or requires `iproc` to continue execution, such as sysreset or monitor, `iproc` will halt execution and exit.

ivxirm

Syntax

ivxirm [-diptvDILMS] [*arguments*...]

Description

The `ivxirm` (the resource manager) initializes the VXI and MXI buses by reading several configuration files and by polling the VXI devices to determine their resources and capabilities. Then, using a set of rules governing VXI configuration, it defines the relationships between commanders and servants and writes this information to the `rsrcmgr.out` configuration file. The resource manager also optionally prints this information to the standard output. The resource manager is usually run automatically at system power-on.

The command line argument definitions follow:

`d`	The next argument contains the name of the directory for the static and operating configuration files. This defaults to `/etc/opt/sicl/vxi`*LU*. Where *LU* is the logical unit number of the VXI interface.
`i`	Ignore static configuration files. The static configuration files contain a set of rules for the resource manager to use during configuration. With this option, the resource manager ignores the static configuration files and follows only the standard VXI configuration rules.
`p`	Print the results of the configuration using the `ivxisc` program.
`t` *time*	Delay the seconds specified in *time* before starting. The recommended time is five seconds. The VXI Standard requires these five seconds to allow instruments to complete their self test. The default is no delay at all.
`v`	Print a verbose output of the resource manager's actions. This is useful for debugging the cardcage configuration.
`D`	The next argument specifies the directory that contains the `ivxisc` program. This defaults to `/opt/sicl/bin`.

`I`	The next argument contains the name of the VXI interface that the resource manager will use to access the VXI bus. This argument is provided mainly for controllers which can connect to multiple, separate VXI systems through multiple VXI or MXI interfaces. This defaults to `vxi`.
`L`	Send all messages to a file named `rsrcmgr.err` in the directory for static and operating configuration files.
`M`	Set the limits for allocation of A24 and A32 memory space to the maximum addresses for that space. The default limits will be set so that the upper and lower one-eighth of A24 and A32 space will not be allocated.
`S`	The next argument contains the name of the program to use to print the VXI configuration. This defaults to the `ivxisc` program

The resource manager first accesses the configuration files as directed by the argument above. It then determines resource and capability information from the VXI devices in the cardcage or multi-cardcage hierarchy. The resource manager then determines the proper configuration according to the rules defined by the configuration files and the standard VXI configuration methods. It then sends appropriate commands to the VXI devices. The configuration is optionally printed. Finally, the configuration information is stored in the rsrcmgr.out file for use by other programs. The `rsrcmgr.out` file contains binary data, not ASCII text.

In the case of multiframe (extended) VXI systems using VXI-MXI bus extenders, the resource manager will set up logical address windows, A16/A24/A32 windows, and interrupt routing registers prior to establishing the commander-servant hierarchy and initiating normal operation.

The VXI/MXI configuration files specify the site-dependent configuration rule changes. See "The VXI/MXI Configuration Files" earlier in this appendix for a description of the file contents.

Note `ivxirm` is normally run automatically from the `iproc` daemon. It cannot be run a second time (manually) without asserting the VXI SYSRESET (`iclear` command) or cycling the mainframe power.

Example

```
ivxirm -p
```

ivxisc

Syntax

ivxisc [-sdvfphmi] [*directory***]**

Description

The `ivxisc` command reads the operating configuration file, `/etc/opt/sicl/vxi`*LU*`/rsrcmgr.out` (where *LU* is the logical unit of the VXI/MXI interface) and prints a human readable display of the current configuration. This display includes slot number tables for each VXI bus in the configuration and logical address tables for each MXI bus, a device table, VME device information, a list of failed devices, a protocol support table, the commander servant hierarchy, an A24/A32 memory map and an interrupt line allocation table.

The default command (no arguments) prints all tables.

Parameters:

`s`	Prints bus/slot tables.
`d`	Prints device table.
`v`	Prints VME device table.
`f`	Prints failed device table.
`p`	Prints protocol table.
`h`	Prints hierarchy.
`m`	Prints memory map.
`i`	Prints IRQ table.
directory	Operating file directory. (default: `/etc/opt/sicl/vxi`*LU*)

Examples

For the VXI interface at logical unit (LU) 16:

```
ivxisc /etc/opt/sicl/vxi16
```

A sample output follows.

```
VXI Current Configuration:

VXI Bus: 0
   Device Logical Addresses:  0  127
Slots:           0  1  2  3  4  5  6  7  8  9 10 11 12
```

```
                -- -- -- -- -- -- -- -- -- -- -- -- --
Empty               O  O  O  O  O     O  O  O  O  O  O
Single Device    X                 X
Multiple Devices
VME
Failed

MXI Bus: 127
   Device Logical Addresses:  127  130

VXI Bus: 130
   Device Logical Addresses:  130  136  145  147
Slots:           0  1  2  3  4  5  6  7  8  9 10 11 12
                -- -- -- -- -- -- -- -- -- -- -- -- --
Empty               O           O  O  O  O  O  O  O  O
Single Device    X     X  X  X
Multiple Devices
VME
Failed


VXI Device Table:

Name      LADD Slot Bus Manufacturer      Model
----      ---- ---- --- ------------      -----
v700ctlr  0    0    0 Hewlett Packard  E1497 Series 700 Controller
vximxi    127    6    0 National Instrum  VXI-MXI Extender
hpvximxi  130    *  127 Hewlett Packard   E1482 VXI-MXI Extender
pwrmeter  136    3  130 Hewlett Packard   E1416 Power Meter
dev1      145    2  130 Racal Dana        0xfff0
dvm       147    4  130 Hewlett Packard   E1326 5 1/2 digit DVM

* - MXI device
```

ivxisc Output example (cont.)

```
VME Device Table:

Name          Bus  Slot  Space  Size
----          ---  ----  -----  ----
No VME cards configured.


Failed Devices:
Name          Bus  Slot  Manufacturer      Model
----          ---  ----  ------------      -----
No FAILED devices detected.


Protocol Support (Msg Based Devices):

Name   CMDR SIG MSTR INT FHS SMP RG EG ERR PI PH TRG I4 I LW ELW 1.3
----    --- --- ---- --- --- --- -- -- --- -- -- --- -- - -- --- ---
v700ctlr X   X   X              X     X      X                 X
pwrmeter        X              X  X  X        X   X X         X
dev1                X                        X     X


Commander/Servant Hierarchy;

    v700ctlr
        pwrmeter
        dev1
        dvm
      vximxi
      hpvximxi

Memory Map:

A24                  Device Name
---                  -----------
0x200000 - 0x23ffff  v700ctlr

A32                      Device Name
---                      -----------
No devices mapped into A32 space.
```

ivxisc Output example (cont.)

```
Interrupt Request Lines:
                  Handler          Interrupter
Name           1 2 3 4 5 6 7    1 2 3 4 5 6 7
----           - - - - - - -    - - - - - - -
v700ctlr       X X X X X X X
vximxi
hpvximxi
pwrmeter
dev1
dvm

VXI-MXI IRQ Routing:

Name           1 2 3 4 5 6 7
----           - - - - - - -
vximxi         I I I I I I I
hpvximxi       O O O O O O O
   I - MXI->VXI
   O - VXI->MXI
   * - Not Routed

VXI-MXI Registers:

Name
----
vximxi
   laddr window register: 0x5b80 range: 128 - 159
   a24   window register: disabled
   a32   window register: disabled
   Interrupt Configuration Register: 0x7f7f
hpvximxi
   laddr window register: 0x7b80 range: 128 - 159
   a24   window register: disabled    a32
   window register: disabled
   Interrupt Configuration Register: 0x7f00
```

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.

Interpreted SCPI
A SICL interface type that allows you to talk to register-based instruments with the high-level SCPI commands.

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).

Index

Symbols

_siclcleanup (C), **387**
“vximesdev.c’’ example, **108**

A

Access modes
  VME, **124**
Active Controller, **74**, **78**
Active Controller, GPIB, **247**
Address
  bus, GPIB, **248**
  cmdr, **85**
  device, **228**
  GPIO symbolic name, **92**
  HP-IB symbolic name, **77**
  interface, **236**
  LAN symbolic name, **169**
  logical unit (lu), **236**
  logical unit (lu) information, **237**
  logical unit (lu) list, **238**
  Primary, **73**
  RS-232 symbolic name, **150**
  Secondary, **73**
  session, **226**
  symbolic name, **40**, **118**
  VXI/MXI symbolic name, **118**
Addressing
  Commander sessions, **41**
  Device sessions, **39**
  GPIO interface sessions, **92**
  HP-IB commander sessions, **85**
  HP-IB device sessions, **72**
  Interface sessions, **40**
  LAN-gatewayed sessions, **169**
  Parallel interface sessions, **92**
  RS-232 device sessions, **146**
  RS-232 interface sessions, **150**
  serial device sessions, **146**
  serial interface sessions, **150**
  VXI/MXI interface sessions, **118**
  VXI/MXI message-based device sessions, **106**
  VXI/MXI register-based device sessions, **110**
Archive libraries, **400**
Argument modifier, **46**
Array size, **46**
Asynchronous Events
  disable, **271**
  enable, **272**
Asynchronous events, **54**
  Interrupts, **55**
  SIGUSR2, **57**
  SRQs, **54**
ATN, See GPIB lines
Attention (ATN) Line, See GPIB

B

Bad address, **197**
Baud Rate, **330**
Big-endian Byte Order, **355**
Block Transfers, **213**
  from FIFO, **300**
  to FIFO, **313**
BREAK, **333**
BREAK, sending, **329**
Buffers
  data structure, **342**
  flush, **221**
  Flushing, **50**
  set size, **340**
  set size and location, **353**
Building SICL applications on HP-UX 10, **399**
Bus Address, GPIB, **248**
Bus errors, **117**
  Example, **117**
Byte Order
  big-endian, **355**
  determine, **356**
  little-endian, **355**

C

Clear
  device, **218**
  interface, **218**
cmdr string, **41**
  HP-IB, **85**
  LAN, **169**

cmdrsrvt.cf file, **416**
Comma operator, **46**
Command Module, **104**
  communicating, **109**
Commander
  close, **219**
  interrupts, **344**
  lock, **279**
  session, **295**
  set status byte, **352**
Commander servant hierarchy, **416**
Commander sessions, **41**
  Addressing, **41**
  HP-IB addressing, **85**
  HP-IB communicating, **85**
  LAN addressing, **169**
Commands
  Word-serial, **130**
Communication sessions, **38**
  GPIO, **91**
  HP-IB, **71**
  LAN, **169**
  Paralle, **91**
  RS-232, **145**
  Serial, **145**
  VXI/MXI, **103**
Compile errors
  Troubleshooting, **192**
  Unexpected symbol, **192**
Compile/Link errors
  Undefined id, **193**
Compiling, **29**
Configuration
  LAN, **168**
  VXI/MXI system, **410**
  VXI/MXI Utilities, **422**
Configuration files, **413**
  cmdrsrvt.cf, **416**
  dynamic.cf, **414**
  irq.cf, **415**
  names.cf, **416**
  ttltrig.cf, **417**
  vmedev.cf, **415**
  vximanuf.cf, **414**
  vximodel.cf, **414**
Connection refused (LAN), **201**
Conversion Characters, **308**, **324**
Conversion characters
  iprintf, **47**
  iscanf, **47**
C-SCPI, **104**
  communicating, **109**

D

D32
  32-bit access, **114**
Data Transfer
  direct memory access (DMA), **269**
  interrupt driven (INTR), **269**
  polling mode (POLL), **269**
  set preferred mode, **269**
DAV, See GPIB lines
Device
  address, **228**
  clear, **218**
  close, **219**
  disable front panel, **318**
  get device address, **228**
  get interface of, **233**
  interrupts, **343**
  lock, **279**
  remote mode, **318**
  session, **294**, **295**
  status byte, **317**
  unlock, **361**
Device sessions, **39**
  Addressing, **39**
  HP-IB, **72**
  HP-IB addressing, **72**
  HP-IB example, **75**
  LAN communicating, **169**
  LAN-gatewayed, **169**
  LAN-gatewayed addressing, **169**
  RS-232, **146**
  RS-232 addressing, **146**
  RS-232 example, **149**
  Serial, **146**
  VME devices, **121**
  VXI/MXI, **103**
  VXI/MXI addressing, **106**, **110**

VXI/MXI communicating, 104
VXI/MXI example, 108, 114
VXI/MXI register programming, 112
Disable Asynchronous Event Handlers, 271
Disable events, 55
DMA, 270
DMA transfers
VXI, 135
Documentation
GPIO Interface, 23
HP-UX, 23
LAN, 24
LAN/HP-IB Gateway, 24
Series 700 Computer, 23
Series 700 RS-232, 23
SICL, 22
VXI/MXI, 24
dynamic.cf file, 414
Dynamically configured devices, 414

E

E1482
and VME, 122
Enable
Error handler, 61
Events, 54, 55
Interrupt events, 55
SRQ handlers, 54
Enable Asynchronous Event Handlers, 272
END Indicator, 224, 225, 260, 268, 316, 357, 383
using with iscanf, 319
END indicator, 50
EOI, See GPIB lines
errhand.c example, 62
Error codes, 190
Error handlers, 61
Creating your own, 63
Example, 62
Troubleshooting, 189
Error messages, 190
Error routines, 61
I_ERROR_EXIT, 61
I_ERROR_NO_EXIT, 61
Errors
Codes, 190
current handler setting, 239
get error code, 229
get error message, 231
handlers, 288
multiple threads, 217, 230, 288
simulate, 217
Troubleshooting, 192
Events
Asynchronous, 54
Disable, 55
Enable, 54, 55
Interrupts, 55
SRQs, 54
Events, see Asynchronous Events
Examples
“vximesdev.c”, 108
Catching bus errors, 117
errhand.c, 62
formatio.c, 49
gpiointr.c, 96
gpiomeas.c, 95
hpibdev.c, 75
hpibintr.c, 81
hpibstatus.c, 79
idn.c, 27
interrupts.c, 59
locking.c, 67
nonformatio.c, 53
serialdev.c, 149
serialintr.c, 154
vmeintr, 127
vmeintr.c, 140
vxiintr.c, 120
vxiregdev.c, 114
Executing a program, 31

F

Features
SICL, 20
Field width, 45
FIFO Transfers, 300, 313
File structure, 390

SICL-DIAG, 396
SICL-GPIO, 392
SICL-HPIB, 392
SICL-LAN, 393
SICL-LANSVR, 393
SICL-MAN, 394
SICL-MAN-GPIO, 394
SICL-MAN-HPIB, 394
SICL-MAN-LAN, 395
SICL-MAN-LANSVR, 396
SICL-MAN-RS232, 395
SICL-MAN-VXI, 395
SICL-PRG, 391
SICL-RS232, 393
SICL-RUN, 390
SICL-VXI, 391
Flow Control, 331
Flushing buffers, 50
Format string, 50
formatio.c example, 49
Formatted Data
read, 223, 312, 319
read format conversion characters, 324
read format modifiers, 322
read white-space, 321
set buffer size, 340
set buffer size and location, 353
write, 225, 302, 312
write format conversion characters, 308
write format flags, 306
write format modifiers, 304
write special characters, 303
Formatted I/O, 42
Argument modifier, 46
Array size, 46
Buffers, 50
Comma operator, 46
Conversion, 43
Example, 49
Field width, 45
Format string, 50
iprintf conversion characters, 47
iscanf conversion characters, 47
Precision, 45
Functions
GPIO specific, 98
HP-IB specific, 88
iabort, 212
iclear, 93, 148, 151
ionsrq, 148, 152
iprintf, 93, 148
ipromptf, 148
iread, 93, 151
ireadstb, 94, 148
iscanf, 93, 148
iserialmclctrl, 153
iserialmclstat, 153
iserialstat, 153
itermchr, 93
itrigger, 93, 148, 151
iwrite, 93, 151
ixtrig, 93, 151
LAN specific, 186
RS-232 specific, 156
VXI/MXI specific, 141

G

Gateways
LAN sessions, 169
GET
HP-IB interface sessions, 78
GPIB, 77, 253
active controller, 247
Addressing commander sessions, 85
Addressing device sessions, 72
ATN (Attention) line control, 245
bus address, 248, 251
bus lines, 248
byte order of data, 356
change bus address, 246
Communicating with commanders, 85
Communicating with interfaces, 77
Device sessions, 72
functions, see igpib*
interface status, 247
interrupts, 343, 344

lines
active controller, 248
ATN (Attention), 248
DAV (Data Valid), 248
EOI (END or Identify), 248
IFC (Interface Clear), 248
listener, 248
LLO (Local Lockout), 248
NDAC (Not Data Accepted), 248
NRFD (Not Ready for Data), 248
REM (Remote), 248
REN (Remote Enable), 248
SRQ (Service Request), 248
talker, 248
listener, 247
local lockout, 250
not data accepted (NDAC), 247
parallel poll, 252, 254
pass control, 251
remote enable, 248, 255
remote mode, 247, 318
send commands, 256
service requests (SRQ), 247
status, 247
system controller, 247
t1 delay, 249, 257
talker, 247
triggers, 359, 385
GPIO
auto-handshake, 259
auto-handshake status, 267
auxiliary control lines, 259
control lines, 260
control lines status, 267
data width, 263, 264
data-in clocking, 262
data-in line status, 267
data-out lines, 260
END pattern matching, 260, 268
enhanced mode status, 268
external interrupt request (EIR) status, 267
functions, see igpio*
handshake status, 267
interface control, 258
interface line polarity, 261
Interface sessions, 92
interface status, 267
interrupts, 343, 344, 346
PCTL delay value, 261
peripheral control (PCTL) line, 260
peripheral status (PSTS) line, 259, 267, 268
SICL functions, GPIO specific, 98
status, 266
status lines, 268
triggers, 359, 385
GPIO interface manuals, 23
gpiointr.c example, 96
gpiomeas.c example, 95

H

Handlers
enable asynchronous event handlers, 272
Error, 61
error, 288
error handler setting, 239
Interrupt, 55
interrupt, 291
interrupt handler address, 240
remove interrupt handler, 292
remove SRQ handler, 293
service request (SRQ), 293
SRQ, 54
SRQ handler address, 241
timeout, 381
Wait for, 56
wait for, 381
Header file
sicl.h, 37
Help
Online, 32
Hostname

LAN, 169
HP-IB, 77
Addressing commander sessions, 85
Addressing device sessions, 72
Communicating with commanders, 85
Communicating with interfaces, 77
Device session example, 75
Device sessions, 72
Interface session example, 79, 81
Primary address, 73
Secondary address, 73
SICL functions, 88
HP-IB commander sessions
iread, 87
ireadstb, 87
iwrite, 87
HP-IB device sessions
iclear, 74
Interrupts, 74
iread, 74
ireadstb, 74
itrigger, 74
iwrite, 74
Service requests, 74
HP-IB interface manuals, 23
HP-IB interface sessions
iclear, 78
Interrupts, 78
iread, 78
itrigger, 78
iwrite, 78
ixtrig, 78
Service requests, 79
HP-IB, See GPIB
hpibdev.c example, 75
hpibintr.c example, 81
hpibstatus.c example, 79
HP-UX
Signals, 57
HP-UX 9
Updating SICL to 10, 399
HP-UX manuals, 23
Hung LAN server, 200

I

I/O operation timesout, 198
I_ERR_NOLOCK, 64
I_ERROR_EXIT, 61
I_ERROR_NO_EXIT, 61
I_ORDER_BE, 356
I_ORDER_LE, 356
iabort, 212
LAN interface sessions, 177
ibblockcopy, 213
iblockcopy, 213
iblockmovex, 215
ibpeek, 229, 296
ibpoke, 229, 298
ibpopfifo, 300
ibpushfifo, 313
icauseerr, 217
iclear, 148, 151, 218, 333
GPIO interface sessions, 93
HP-IB device sessions, 74
HP-IB interface sessions, 78
LAN interface sessions, 177
VXI/MXI device sessions, 130
VXI/MXI interface sessions, 131
iclear utility, 403
iclose, 28, 219
iderefptr, 220
idn.c example, 27
idrvrversion
LAN-gatewayed sessions, 174
IEEE-488, See GPIB
IFC
HP-IB interface sessions, 78
IFC, See GPIB lines
iflush, 221, 225
ifread, 52, 223
termination character, 357
ifwrite, 52, 225
igetaddr, 226
igetdata, 227
igetdevaddr, 228
igeterrorno, 229
igeterrstr, 231
igetgatewaytype, 232
igetintfsess, 233

igetintftype, **234**
igetlockwait, **235**
igetlu, **236**
igetluinfo, **237**
  LAN interface sessions, **177**
igetlulist, **238**
igetonerror, **239**
igetonintr, **240**
igetonsrq, **241**
igetsesstype, **242**
igettermchr, **243**
igettimeout, **178**, **244**
igpibatnctl, **245**
igpibbusaddr, **246**
igpibbusstatus, **247**
igpibgett1delay, **249**
igpibllo, **250**
igpibpassctl, **251**
igpibppoll, **252**
igpibppollconfig, **253**
igpibppollresp, **254**
igpibrenctl, **255**
igpibsendcmd, **256**, **359**
igpibsett1delay, **257**
igpioctrl, **258**
ihint, **269**
iintroff, **55**, **271**
iintron, **55**, **272**
ilangettimeout, **273**
ilantimeout, **178**, **274**
ilblockcopy, **213**
ilocal, **277**
ilock, **64**, **278**
ilpeek, **229**, **296**
ilpoke, **229**, **298**
ilpopfifo, **300**
ilpushfifo, **313**
imap, **112**, **229**, **280**
imapinfo, **282**, **285**, **286**
imapx, **283**
INST, **28**
Interface
  address, **236**
  clear, **218**
  close, **219**
  get type of, **234**
  interrupts, **343**
  lock, **278**
  logical unit (lu) information, **237**
  logical unit (lu) list, **238**
  serial status, **336**
  session, **233**, **294**, **295**
  set up serial characteristics, **330**
  unlock, **361**
Interface sessions, **40**
  Addressing, **40**
  GPIO, **92**
  GPIO addressing, **92**
  GPIO example, **95**
  HP-IB communicating, **77**
  HP-IB example, **79**, **81**
  LAN, **176**
  Parallel, **92**
  RS-232, **150**
  RS-232 addressing, **150**
  RS-232 communicating, **150**
  RS-232 example, **154**
  Serial, **150**
  VXI/MXI, **103**
  VXI/MXI addressing, **118**
  VXI/MXI communicating, **118**
  VXI/MXI example, **120**
Interrupt handlers, **55**
  Example, **59**
Interrupts
  commander-specific, **345**, **347**, **348**, **350**
  data transfer, **270**
  device-specific, **344**, **346**, **347**, **349**
  enable and disable, **343**
  GPIB, **344**
  GPIO, **346**
  GPIO example, **96**
  GPIO interface sessions, **94**
  handler, **291**
  handler address, **240**
  HP-IB device sessions, **74**
  HP-IB interface sessions, **78**
  interface-specific, **344**, **346**, **347**, **349**
  multiple threads, **382**

nesting, 272
RS-232 device sessions, 148
RS-232 interface sessions, 153
serial (RS-232), 347
set for commander session, 344
set for device session, 343
set for interface session, 343
VME, 125
VXI, 349
VXI/MXI, 138
interrupts.c example, 59
Invalid address, 194
Invalid INST, 194
ionerror, 61, 288
ionintr, 55, 291
LAN interface sessions, 177
ionsrq, 54, 148, 152, 293
LAN interface sessions, 177
iopen, 28, 226, 229, 294
iopen fails, 194, 197, 198
IP address, 169
ipeek, 114, 125, 296
ipeek utility, 405
ipeek16x, 297
ipeek32x, 297
ipeek8x, 297
ipoke, 114, 125, 298
ipoke utility, 406
ipoke16x, 299
ipoke32x, 299
ipoke8x, 299
ipopfifo, 300
iprintf, 42, 93, 148, 225, 229, 302
conversion characters, 47
iproc utility, 423
ipromptf, 42, 148, 229, 312
ipushfifo, 313
iread, 52, 151, 224, 315
GPIO interface sessions, 93
HP-IB commander sessions, 87
HP-IB device sessions, 74
HP-IB interface sessions, 78
LAN-gatewayed sessions, 174
termination character, 357
VXI/MXI device sessions, 130
iread utility, 407
ireadstb, 94, 148, 317
HP-IB commander sessions, 87
HP-IB device sessions, 74
VXI/MXI device sessions, 130
iremote, 318
IRQ lines, 415
irq.cf, 415
iscanf, 42, 93, 148, 223, 229, 319
conversion characters, 47
notes on using, 319
using with itermchr, 319
I-SCPI
communicating, 109
iserialbreak, 329
iserialctrl, 330
iserialmclctrl, 153, 334
iserialmclstat, 335
iserialstat, 153, 336
isetbuf, 340
isetdata, 227, 342
isetintr, 55, 343
isetlockwait, 278, 351
isetstb, 352
isetubuf, 353
isprintf, 229, 302
isscanf, 229, 319
isvprintf, 229, 302
isvscanf, 229, 319
itermchr, 93, 224, 316, 357
using with iscanf, 319
itimeout, 28, 358
itrigger, 133, 148, 151, 359
GPIO interface sessions, 93
HP-IB device sessions, 74
HP-IB interface sessions, 78
VXI/MXI device sessions, 130
iunlock, 64, 361
iunmap, 114, 125, 282, 285, 362
iversion, 365
ivprintf, 229, 302
ivpromptf, 229, 312
ivscanf, 229, 319
ivxibusstatus, 366
ivxigettrigroute, 369

ivxirm utility, **425**
ivxirminfo, **370**
ivxisc, **421**
ivxisc utility, **428**
ivxiservants, **372**
ivxitrigoff, **373**
ivxitrigon, **375**
ivxitrigroute, **360**, **377**, **386**
ivxiwaitnormop, **379**
ivxiws, **380**
iwaithdlr, **56**, **381**
iwblockcopy, **213**
iwpeek, **229**, **296**
iwpoke, **229**, **298**
iwpopfifo, **300**
iwpushfifo, **313**
iwrite, **52**, **151**, **225**, **383**
  GPIO interface sessions, **93**
  HP-IB commander sessions, **87**
  HP-IB device sessions, **74**
  HP-IB interface sessions, **78**
  LAN-gatewayed sessions, **174**
  VXI/MXI device sessions, **130**
  VXI/MXI interface sessions, **131**
iwrite utility, **408**
ixtrig, **151**, **359**, **360**, **384**
  GPIO interface sessions, **93**
  HP-IB interface sessions, **78**

L

LAN
  Addressing LAN-gatewayed sessions, **169**
  client/server, **163**
  Communication sessions, **169**
  Configuration, **168**
  get gateway type, **232**
  hostname, **169**
  interface lock not supported, **278**
  Interface sessions, **176**
  IP address, **169**
  networking protocols, **166**
  Overview, **163**
  Performance, **168**
  Servers, **167**
  set timeout, **274**
  SICL functions, **186**
  SICL LAN Protocol, **166**
  Signal handling, **184**
  software architecture, **165**
  TCP/IP Instrument Protocol, **166**, **172**
  timeout value, **273**
  Timeouts, **178**
  timeouts with multiple threads, **276**
  Troubleshooting, **195**
LAN client
  definition, **163**
  LAN-gatewayed sessions, **169**
  Troubleshooting, **195**, **197**
LAN error
  Bad address, **197**
  Connection refused, **201**
  I/O operation timesout, **198**
  iopen fails, **197**, **198**
  portmapper, **201**
  RPC system error, **201**
  syntax erro, **197**
  Unrecognized symbolic name, **197**
LAN interface sessions
  iabort, **177**
  iclear, **177**
  igetluinfo, **177**
  ionintr, **177**
  ionsrq, **177**
LAN manuals, **24**
LAN server
  Appears hung, **200**
  definition, **163**
  Description of, **167**
  LAN-gatewayed sessions, **169**
  portmapper error, **201**
  RPC system error, **201**
  siclland daemon, **199**
  Troubleshooting, **195**, **199**
LAN/HP-IB Gateway manual, **24**
LAN-gatewayed sessions, **169**
  idrvrversion, **174**
  iread, **174**
  iwrite, **174**

LAN-to-Instrument Gateway, 164
Link errors
  Troubleshooting, 192
Linking, 29
  Archive libraries, 400
Listener, GPIB, 247
Little-endian Byte Order, 355
LLO, See GPIB lines
Local Lockout, GPIB, 248, 250
Local Mode, 277
Lock, 278
  commander, 279
  device, 279
  hangs due to, 278
  interface, 278
  nesting, 279, 361
  unlock, 361
  wait status, 351
Lock actions, 65
Locking, 64
  Example, 67
Locking Multi-user environment, 66
locking.c example, 67
Lockwait Flag Status, 235
Logical Unit
  address, 236
  information, 237
  list, 238

M

man pages, 32
Manuals
  GPIO Interface, 23
  HP-IB Interface, 23
  HP-UX, 23
  LAN, 24
  LAN/HP-IB Gateway, 24
  Series 700 Computer, 23
  Series 700 RS-232, 23
  SICL, 22
  VXI/MXI, 24
Manufacturer id, 414
Map
  memory, 280, 283
Mapping memory
  32-bit access, 114
  Register-based devices, 112
  VEM devices, 126
Memory
  get hardware contraint information, 286
  hardware constraints, 282, 285
  map, 280, 283
  read, 296, 297
  unmap, 362
  write, 298, 299
Memory mapping, 112
Memory space
  Mapping, 114, 125
Message-Based devices, 104, 130
  communicating, 105
Message-based programming
  Example, 108
Messages
  Error, 190
Modem Control Lines, 334
Move data, Data, move, 215
Multi-user environment
  Locking, 66
MXI, 368
  Triggering, 417

N

names.cf file, 416
NDAC, See GPIB
Nesting
  interrupts, 272
  locks, 279, 361
Networking Protocols, 166
Networking, see LAN
Newline character, 50
nonformatio.c example, 53
Non-formatted I/O, 52
  Example, 53
Normal Operation (VXI), 379
Notification
  Interrupts, 55
NRFD, See GPIB lines

O

Online help, 32
Opening a session, 38
oride.cf file
  and VME, 122
Overview
  SICL, 20
  VXI/MXI configuration, 411

P

Parallel
  Interface sessions, 92
  SICL functions, parallel specific, 98
parallel poll, 253
Parallel Poll, GPIB, 252, 253, 254
Parity, 330
Pass Control, 78
Pass Control, GPIB, 251
Performance
  LAN, 168
Polling, 270
portmapper, 201
Precision, 45
Preference for Data Transfer, 269
Primary address, 73, 106, 110
Program hangs, 194
Programming
  Register, 112
Protocols, Networking, 166

R

Read
  buffered data, 223
  formatted data, 312, 319
  memory, 296, 297
  unformatted data, 315
Register programming, 112, 114, 125
  Catching bus errors, 117
  Example, 114
Register-Based devices, 104, 130
  communicating, 109
  Mapping memory space, 112
REM, See GPIB lines
Remote Enable, 248
Remote Enable, GPIB, 255
Remote Mode, 248, 318
Remote Mode, GPIB, 247
REN, See GPIB lines
Resource Manager, 410, 412, 425
Resource Manager (VXI), 370
Resources
  Declaring VME, 122
Routines Formatted I/O
  Routines, 51
rpcinfo troubleshooting, 199, 201
RS-232
  Device sessions, 146
  Interface sessions, 150
  Interrupts, 148, 152
  Service requests (SRQs), 152
  SICL functions, RS-232 specific, 156
RS-232 manuals, 23
RS-232, see Serial
Running a program, 31
Run-time errors
  Invalid address, 194
  Invalid INST, 194
  iopen fails, 194
  Program hangs, 194
  Timeout occurred, 194
  Troubleshooting, 194

S

SCPI, 104
Secondary address, 73
Send Commands, GPIB, 256
Serial
  baud rate, 330
  Device sessions, 146
  END Indicator for read, 331
  END Indicator for write, 333
  flow control, 331
  functions, see iserial*
  Interface sessions, 150
  interface status, 336
  Interrupts, 148, 152
  interrupts, 344, 347
  modem control lines, setting, 334
  modem control lines, status, 335

parity, 330
resetting interface, 333
sending BREAK, 329
Service requests (SRQs), 152
set up interface, 330
SICL functions, serial specific, 156
stop bits, 330
triggers, 359, 385
serialdev.c example, 149
serialintr.c example, 154
Series 700 Computer manuals, 23
Series 700 RS-232 manuals, 23
Servant Area (VXI), 366
Servants (VXI), 372
Servers
LAN, 167
Service request
HP-IB device sessions, 74
HP-IB interface sessions, 79
Service Requests (SRQs), 241, 247
handlers, 293
Session
close, 219
commander, 295
data structure, 227, 342
device, 294, 295
get address of, 226
get type, 242
interface, 294, 295
open, 294
Sessions, 38
Addressing GPIO interfaces, 92
Addressing HP-IB commanders, 85
Addressing HP-IB devices, 72
Addressing LAN-gatewayed, 169
Addressing RS-232 devices, 146
Addressing RS-232 Interfaces, 150
Addressing VXI/MXI interfaces, 118
Addressing VXI/MXI message-based devices, 106
Addressing VXI/MXI register-based devices, 110
Commander, 41
Device, 39
GPIO interface, 92
HP-IB, 71
HP-IB device, 72
Interface, 40
LAN, 169
LAN interface sessions, 176
LAN-gatewayed sessions, 169
Opening, 38
Parallel interface, 92
RS-232 device, 146
RS-232 interface, 150
Serial device, 146
Serial interface, 150
VXI/MXI, 103
VXI/MXI device, 103
VXI/MXI interface, 103
Shared libraries, 29
SICL
Building HP-UX 9 applications, 399
Documentation, 22
Features, 20
File structure, 390
Overview, 20
User, 21
Utilities, 402
SICL LAN Networking Protocol, 166
sicl.h, 28, 37
sicl_tl script, 399
_siclcleanup (C), 387
SICL-DIAG file structure, 396
SICL-GPIO file structure, 392
SICL-HPIB file structure, 392
SICL-LAN file structure, 393
siclland daemon, 199
SICL-LANSVR file structure, 393
SICL-MAN file structure, 394
SICL-MAN-GPIO file structure, 394
SICL-MAN-HPIB file structure, 394
SICL-MAN-LAN file structure, 395
SICL-MAN-LANSVR file structure, 396
SICL-MAN-RS232 file structure, 395
SICL-MAN-VXI file structure, 395
SICL-PRG file structure, 391
SICL-RS232 file structure, 393
SICL-RUN File structure, 390

SICL-VXI file structure, 391
Signal handling with LAN, 184
Signals
  HP-UX, 57
SIGUSR2, 57
SRQ handlers, 54
SRQ, See Service Requests
Status
  GPIB, 247
  lock wait, 351
  of lockwait flag, 235
  VXI bus, 366
Status Byte, 317
  set, 352
Stop Bits, 330
Symbolic name, 40, 118, 416
  GPIO, 92
  HP-IB, 77
  LAN, 169
  RS-232, 150
  VXI/MXI, 118
syntax error, 197
System Controller, GPIB, 247
SystemVXI/MXI
  Configuration, 410

T

T1 Delay, GPIB, 249, 257
Talker, GPIB, 247
TCP/IP Instrument Networking
    Protocol, 166, 172
Termination Character, 224, 316, 357
  get, 243
Threads
  error handling, 288
  errors, 217, 230
  interrupt handling, 382
  LAN timeout, 276
Timeout occurred, 194
Timeouts, 381
  get current value, 244
  LAN, 178
  set wait time, 358
Transfer Blocks, 213
  from FIFO, 300
  to FIFO, 313
Trigger lines
  VXI controller, 132
Triggers
  get VXI trigger information, 369
  GPIB, 385
  GPIO, 385
  send, 359
  send extended trigger, 384
  serial (RS-232), 385
  VXI, 386
  VXI lines status, 366
  VXI, assert, 375
  VXI, de-assert, 373
  VXI, route lines, 377
Troubleshooting
  Compile errors, 192
  Install error handler, 189
  LAN, 195
  LAN client, 197
  LAN server, 199
  Link errors, 192
  Run-time errors, 194
TTL trigger lines
  ttltrig.cf, 417
ttltrig.cf file, 417

U

Undefined id, 193
Unexpected symbol, 192
Unformatted Data
  read, 315
  write, 383
Unlock
  device, 361
  interface, 361
  nesting, 361
Unmap Memory, 362
Unmapping memory space, 114, 125
Unrecognized symbolic name, 197
Using Timeouts with LAN, 178
Utilities
  iclear, 403
  ipeek, 405
  ipoke, 406

iproc, 423
iread, 407
ivxirm, 425
ivxisc, 428
iwrite, 408
VXI/MXI, 422

V

Version, of SICL Software, 365
VME
Access modes, 124
and the E1482, 122
Communicating with devices, 121
Declaring Resources, 122
Example program, 126
Interrupts, 125
oride.cf file, 122
VME devices, 415
Example of programming, 127
VME interrupts
Example, 140
vmedev.cf, 415
Example, 126
vmedev.cf file, 122
vmeintr.c example, 127, 140
VXI
bus status, 366
DMA transfers, 135
information structure, 370
interrupts, 349
normal operation, 379
resource manager, 370
send word-serial commands, 380
servant area, 366
servants, list of, 372
trigger lines, 366
trigger, assert, 375
trigger, de-assert, 373
trigger, route lines, 377
triggers, 360, 386
VXI controller trigger lines, 132
VXI, get trigger information, 369
VXI/MXI
Addressing interfaces sessions, 118
Addressing message-based device session, 106
Addressing register-based device session, 110
Communication sessions, 103
Configuration, 410, 412
Configuration files, 413
Configuration Utilities, 422
Interrupts, 138
IRQ lines, 415
Mapping memory space, 112
Message-Based devices, 104, 130
message-based programming example, 108
Overview of configuration, 411
Register programming, 112
Register programming example, 114
Register-Based devices, 104
Resource Manager, 412
SICL functions, 141
VME devices, 121
VXI/MXI device sessions, 103
Example, 108, 114
iclear, 130
ionsrq, 130
iread, 130
ireadstb, 130
itrigger, 130
iwrite, 130
VXI/MXI interface sessions, 103
Example, 120
iclear, 131
ireadiread
VXI/MXI interface sessions, 131
iwrite, 131
VXI/MXI manuals, 24
VXI/MXI model number, 414
VXI/MXI Register-Based devices, 130
VXI/MXI trigger lines, 132
vxiinfo Structure, 370
vxiintr.c example, 120
vximanuf.cf file, 414
vximodel.cf file, 414
vxiregdev.c example, 114

W

Wait
  for handlers, **56**, **381**
  for normal VXI operation, **379**
Wait, lock status, **351**
Word-serial Commands (VXI), **380**
Write
  buffered data, **225**
  formatted data, **302**, **312**
  memory, **298**, **299**
  unformatted data, **383**