HEWLETT
PACKARD

User's Guide
for HP-UX

HP Standard Instrument Control Library

HP Standard Instrument Control Library

User's Guide for HP-UX

Notice

Printing History

This is the fifth edition of the *HP Standard Instrument Control Library User's Guide for HP-UX.*

May 1994 — First Edition

September 1994 — Second Edition

January 1995 — Third Edition

November 1995 — Fourth Edition

September 1996 — Fifth Edition

Contents

1 Introduction

Introduction

Welcome to the *HP Standard Instrument Control Library (SICL) User's Guide for HP-UX*. This guide describes how to use SICL to develop I/O applications on the HP-UX version 10.20 (or later) operating system. 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: HP-IB, GPIO, VXI/MXI, RS-232, and LAN.

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

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

- Chapter 2 - **Getting Started with HP SICL** steps you through building and running a simple example program. This is a good place to start if you are a first-time SICL user.
- Chapter 3 - **Using HP 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 HP SICL with HP-IB** describes how to communicate over the HP-IB interface. Example programs are also provided.
- Chapter 5 - **Using HP SICL with GPIO** describes how to communicate over the GPIO interface. Example programs are also provided.
- Chapter 6 - **Using HP SICL with VXI/MXI** describes how to communicate over the VXIbus. Example programs are also provided.
- Chapter 7 - **Using HP SICL with RS-232** describes how to communicate over the RS-232 interface. Example programs are also provided.
- Chapter 8 - **Using HP SICL with LAN** describes how to communicate over a LAN. Example programs are also provided.
- Chapter 9 - **Troubleshooting Your HP SICL Program** describes some of the most common SICL programming problems and provides suggestions to help you solve the problems.

This guide also contains the following appendices:

- **Appendix A - The HP SICL Files** summarizes where the SICL files are located on your system.
- **Appendix B - Updating HP-UX 9 SICL Applications** describes how to update SICL applications that were written on HP-UX version 9.x to work on HP-UX version 10.x.
- **Appendix C - Porting DIL to SICL** provides tips for moving from the Device I/O Library (DIL) to SICL.
- **Appendix D - The HP SICL Utilities** describes the SICL utilities which can be used to read and write to devices or interfaces from the command line.
- **Appendix E - Customizing Your VXI/MXI System** explains 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**.

HP SICL Overview

SICL is a modular instrument communications library that works with a variety of computer architectures, I/O interfaces, and operating systems. I/O 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 you, the programmer, complete control over I/O communications.

Support

SICL is supported on HP-UX version 10.20 with the following interfaces: HP-IB, GPIO, VXI/MXI, RS-232, and LAN.

Users

SICL is intended for instrument I/O and C/C++ programmers who are familiar with the HP-UX operating system.

Other Documentation

HP SICL Documentation

- *HP I/O Libraries Installation and Configuration Guide for HP-UX* explains how to install and configure the HP I/O Libraries, including the HP SICL and the HP VISA (Virtual Instrument Software Architecture) libraries, on HP-UX.
- *HP SICL Reference Manual* provides the function syntax and description of each SICL function.
- *HP SICL Quick Reference Guide for C Programmers* helps you find SICL function syntax information quickly.
- *HP SICL Online Help* is provided in the form of HP-UX manual pages (man pages) and HyperHelp.
- *HP 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.

Related Product Documentation

- HP-UX 10.x Documentation:
  - *Programming on HP-UX*
  - *HP C Programmer's Guide*
  - *HP C++ Programmer's Guide*
  - *HP C/HP-UX Reference Manual*
  - *Installing HP-UX 10*
  - *Managing HP-UX Software with SD-UX*
  - *Systems Administration Tasks Manual*
  - *Moving HP-UX 9 Code and Scripts to 10*
- HP V743 Controller Documentation:
  - *HP V743 VXI Controller Installation Guide*
  - *HP V743 Owner's Guide*
- HP 9000 Series 700 Computer Interface Documentation:
  - *HP E1482 VXI-MXI Bus Extender User's Guide*
  - *HP E1489I MXI Controller Interface for HP 9000 Series 700 Workstation Installation Guide and Overview*
- GPIO Interface Documentation:
  - *HP E2074 GPIO Interface Installation Guide*
- HP-IB Interface Documentation:
  - *HP E2071 and E2070 HP-IB Interface Installation Guide*
  - *Tutorial Description of the Hewlett-Packard Interface Bus (HP-IB)*
- Series 700 RS-232 Interface Documentation:
  - *The RS-232 Solution* by Joe Campbell, SYBEX Publishing
- LAN Documentation:
  - *Networking Overview*
  - *Installing and Administering LAN/9000 Software*
  - *Administering ARPA Services*
- LAN/HP-IB Gateway Documentation:
  - *HP E2050 LAN/HP-IB 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 HP SICL

Getting Started with HP 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 HP SICL Program
- Compiling and Linking an HP SICL Program
- Running an HP SICL Program
- Getting Online Help
- Where to Go Next

If you need additional information on any of the SICL functions, see the *HP SICL Reference Manual*for details.

Reviewing an HP 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 HP-IB instrument for its identification string and print the results.

```
/* idn.c
   The following program uses SICL to query an HP-IB
   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 a SCPI *RST command and prompt for identification 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 the *HP SICL Reference Manual* 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 HP SICL." You can also see the *HP SICL Reference Manual* for specifics about these SICL function calls.

Compiling and Linking an HP 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:

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

- 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."

NOTE

SICL on HP-UX version 10.20 does *not* support 64-bit operations.

Do not compile with the `+PA2.0` compiler flag.

Running an HP SICL Program

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

```
idn
```

When using an HP 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* for information on running `iosetup`.

Getting Online Help

Online help is offered using Bristol Technology's HyperHelp Viewer, or in the form of HP-UX manual pages (**man** pages). You can get help on the following SICL functions:

- SICL function calls
- SICL utilities

Using the HyperHelp Viewer

The Bristol Technology HyperHelp Viewer allows you to view the *HP SICL Reference Manual* online in HP-UX. To start the HyperHelp Viewer with the SICL help file, type the following:

```
hyperhelp /opt/hyperhelp/siclhelp.hlp
```

When you start the Viewer, you can also specify any of the following options:

`-k` *keyword*	Opens the Viewer and searches for the specified *keyword*.
`-p` *partial_keyword*	Opens the Viewer and searches for a specific *partial keyword*.
`-s` *viewmode*	Opens the Viewer in the specified *viewmode*. If `1` is specified as the *viewmode*, then the Viewer is shared by all applications. If `0` is specified, then a separate Viewer is opened for each application (default).
`-display` *display*	Opens the Viewer on the specified *display*.

Using Manual Pages

To use manual pages, type the HP-UX **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 HP SICL." Additionally, you should look at the chapters that describe how to use SICL with your particular I/O interface(s):

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

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

3

Using HP SICL

Using HP 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 Locks

For specific details on SICL function calls, see the *HP SICL Reference Manual*.

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 HP-IB, 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 HP-IB 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 HP-IB 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 HP SICL with HP-IB") 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 (HP-IB, 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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`	HP-IB interface with the symbolic name `hpib`.
`hpib2`	Second HP-IB interface with the symbolic name `hpib2`.

The following example opens an interface session with the HP-IB interface:

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

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

Commander Sessions

The commander session allows you to talk to the interface controller. Typically, the controller is the computer used to communicate with devices on the interface. 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* for information on these values. The following are valid commander addresses:

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

The following is an example of creating a commander session with the HP-IB 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 **iprompf** 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 the *HP SICL Reference Manual* 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 the *HP SICL Reference Manual* for more information on how data is converted under the control of the format string.

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

Format Flags for `iprintf` and `ipromptf`

Format 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 fo **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 HP-IB 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 Routines

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 the *HP SICL Reference Manual* 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 HP-IB 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 the *HP SICL Reference Manual*.

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 HP-UX Signals

NOTE

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

SICL `hpib` and `vxi` interfaces use an HP-UX 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 HP-UX resources.

Protecting I/O Calls Against Interrupts

It is standard HP-UX 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 Locks

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

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

Locks are performed on a per-session (device, interface, or commander) basis. 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 Visual Engineering Environment (HP VEE) and the HP BASIC/UX 700 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 the *HP SICL Reference Manual* 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 the *HP SICL Reference Manual*for a full description). If the **`isetlockwait`** function is called with the **`flag`** parameter set to 0 (zero), the default action is changed. Rather than causing SICL functions to suspend, an error will be returned immediately.

To return to the default action, 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 Locks," remember that an interface session can access a device locked from a device session.) In general, it is not friendly behavior to lock a device at the beginning of an application and unlock it at the end. This can result in deadlock or long waits by others who want to use the resource.

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

Locking 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 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 HP SICL with HP-IB

Using HP SICL with HP-IB

The HP-IB 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 HP-IB are both used in the discussions and examples in this chapter. The HP-IB related SICL functions have the string `GPIB` embedded in the function name.

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

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

This chapter contains the following sections:

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

Creating a Communications Session with HP-IB

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

Communicating with HP-IB 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 HP-IB 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* for information on these values.

The following are example HP-IB 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`, and so forth.

SICL supports both primary and secondary addressing on HP-IB interfaces.

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

NOTE

If you are using an HP-IB Command Module to communicate with VXI devices, the secondary address must be specified to select a specific instrument in the card cage. 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 HP-IB device at bus address 16:

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

HP SICL Function Support with HP-IB Device Sessions

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

`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.

HP-IB Device Session Interrupts

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

HP-IB Device Sessions and Service Requests

HP-IB device sessions support Service Requests (SRQ). On the HP-IB interface, when one device issues an SRQ, the library will inform *all* HP-IB device sessions that have SRQ handlers installed. (See **`ionsrq`** in the *HP SICL Reference Manual* .) This is an artifact of how HP-IB handles the SRQ line. The interface cannot distinguish which device requested service. Therefore, the library acts as if all devices require service. Your SRQ handler can retrieve the device's **status byte** by using the **`ireadstb`** function. 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 HP-IB interface is the Active Controller. Passing control to another HP-IB device causes the interface to lose active control.

HP-IB Device Session Example

The following example illustrates communicating with an HP-IB device session. This example opens two HP-IB communications sessions with VXI devices (through a VXI Command Module). Then a scan list is sent to a switch, and measurements are taken by the multimeter every time a switch is closed.

```
/* 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 HP-IB 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 HP-IB Interfaces

To create an interface session on your HP-IB 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* for information on these values.

The following are example HP-IB 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`, and so forth.

The following example opens a interface session with the HP-IB interface:

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

HP SICL Function Support with HP-IB Interface Sessions

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

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

HP-IB Interface Session Interrupts

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

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

HP-IB Interface Sessions and Service Requests

HP-IB interface sessions support Service Requests (SRQ). On the HP-IB interface, when one device issues an SRQ, the library will inform *all* HP-IB interface sessions that have SRQ handlers installed. (See **`ionsrq`** in the *HP SICL Reference Manual* .) It is good practice to ensure that a device is not requesting service before leaving the SRQ handler. The easiest technique for this is to service all devices from one handler.

HP-IB Interface Session Examples

Checking the Bus Status

The following example program is an ANSI C program that retrieves the HP-IB 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 HP-IB 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 HP-IB Commanders

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

Addressing HP-IB Commanders

To create a commander session on your HP-IB 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* for information on these values.

The following are example HP-IB 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`**, and so forth.

The following example opens a commander session the HP-IB interface:

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

HP SICL Function Support with HP-IB Commander Sessions

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

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

HP-IB Commander Session Interrupts

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

Summary of HP-IB Specific Functions

NOTE

Using these HP-IB 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
igpibllo	Sets bus in Local Lockout Mode
igpibpassctl	Passes active control to specified address
igpibppoll	Performs a parallel poll on the bus
igpibppollconfig	Configures device for PPOLL response
igpibppollresp	Sets PPOLL state
igpibrenctl	Sets or clears the REN line
igpibsendcmd	Sends data with ATN line set
igpibsett1delay	Sets the T1 delay on the GPIB interface

5

Using HP SICL with GPIO

Using HP SICL with GPIO

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

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 HP I/O Libraries installation. See the *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 HP-IB 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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`**, and so forth.

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

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

HP SICL Function Support with GPIO Interface Sessions

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

`iwrite`, `iread`	The *size* parameters for non-formatted I/O functions are always byte counts, regardless of the current data width of the interface.
`iprintf`, `iscanf`	All formatted I/O functions work with GPIO. When formatted I/O is used with 16-bit data widths, the formatting buffers re-assemble the data as a stream of bytes. On 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 the *HP SICL Reference Manual* for details.
`itermchr`	With 16-bit data widths, only the low (least-significant) byte is used.
`ixtrig`	Provides a method of triggering using either the CTL0 or CTL1 control lines. This function pulses the specified control line for approximately 1 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 HP-IB. On an HP-IB interface, a device can request service from the controller by asserting a line on the interface bus. On GPIO, the EIR line is assumed to be the service request line.
`ireadstb`	The *HP SICL Reference Manual* says that **`ireadstb`** is for device sessions only. Since GPIO has no device sessions, **`ireadstb`** is allowed with GPIO interface sessions. The interface status byte has bit 6 set if EIR is asserted; otherwise, the status byte is 0 (zero). This allows normal SRQ programming techniques in GPIO SRQ handlers.

GPIO Interface Session Interrupts

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

GPIO Interface Session Example

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

#include <sicl.h>

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

   /* 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 an interrupt handler and enables EIR interrupts
   - Waits for EIR; invokes the handler for each interrupt
   - Handler checks interrupt cause and 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 *HP SICL Reference Manual*
`I_GPIO_READ_EOI`	END termination pattern	`I_GPIO_EOI_NONE` or 8-bit or 16-bit mask
`I_GPIO_SET_PCTL`	Start PCTL handshake	1

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

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

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

6

Using HP SICL with VXI/MXI

Using HP 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 HP I/O Libraries installation. See the *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 HP SICL Function Support with VXI/MXI
- Using HP SICL Trigger Lines
- Using **`i?blockcopy`** for DMA Transfers with the V743
- Using VXI Specific Interrupts
- Summary of VXI/MXI Specific Functions

For information on the specific SICL function calls, see the *HP SICL Reference Manual.*

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 E, "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.
- **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`, and so forth) is *not* supported over LAN.

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

Other HP Products:

- HP Compiled SCPI - Use the C-SCPI product and program with high-level SCPI commands (achieve higher throughput as well).
- HP 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-HPIB gateway, such as the HP E2050 LAN/HP-IB 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`**, and so forth.

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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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**, and so forth.

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. This method is supported over LAN.
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 HP Products

HP Compiled SCPI	The HP Compiled SCPI product is another programming language that can be used with SICL to program register-based instruments with SCPI commands. Because this product interprets the SCPI commands at compile time, it can be used to achieve high throughput of register-based devices.
HP Command Module	When you use an HP Command Module to communicate with VXI/MXI devices, you are actually communicating over HP-IB. 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 HP SICL with HP-IB" chapter for more details on communicating through a command module.

If you currently have a SICL application that accesses VXI devices by using HP-IB and the HP E1405/06 Command Module, you can port your application to use the `iscpi` interface and directly access the VXI backplane without the use of the Command Module. This can be done by changing the **`iopen`** function to use the `iscpi` interface followed by the device logical address.

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

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.

Contact your local sales representative for information on other HP products that might interpret the high-level SCPI commands for register-based devices.

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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 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`, and so forth.

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");
```

Interpreted SCPI (`iscpi`) Addressing Rules

The simplest way to address a register-based device using the `iscpi` interface is to use the same rules described in the last section: Specify the interface logical unit or symbolic name and a particular device logical address in the *addr* parameter of the `iopen` function. For example:

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

In most cases this is sufficient and additional addressing is not needed. I-SCPI automatically configures your system according to specific combining rules that determine how the instruments are set up relative to other VXI instruments.

Generally, when an `iopen` is performed, an instrument is formed consisting of all devices at logical addresses contiguous to the base logical address passed in the address string. Let's say, for example, that you open an instrument at logical address 24 and the next logical address is 25. The `iscpi` interface will search for an instrument driver that supports the combined instruments found.

If you wish to specify how instruments are combined or what instrument driver to use, see the following sections for details on specifying this information.

Defining an Instrument. There may be times you would like to have control over which logical addresses are used to form a particular instrument. In this case you can use an explicit list in the logical address portion of the `iopen` call. Define the instrument by adding a colon after the interface symbolic name followed by the backplane name specified in the `iscpi` hwconfig.cf entry (backplane is the *symname* of the VXI backplane SICL driver, usually vxi). Then add the instrument logical addresses enclosed within parentheses separated by commas. For example:

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

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

Defining an Instrument Driver. There may be times when you would like to specify an instrument driver to use for a particular set of logical addresses. This allows you to create your own instrument drivers or you can form unique virtual instrument combinations. This can be done by adding the instrument driver name within brackets. For example:

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

If you would like to specify the instrument driver plus which instruments are grouped together to form the instrument, use the following form:

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

The directory location specified during the SICL interface configuration is searched (default is `/opt/sicl/lib/iscpi`) for a matching instrument driver. Note that the driver name is case sensitive.

NOTE

The `iopen` call will run faster if you specify an instrument driver name since it does not have to search through all the instrument drivers for a match.

Programming with Interpreted SCPI (the `iscpi` Interface)

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

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

The `iscpi` interface was designed to closely simulate control of register-based instruments using the HP Command Module over HP-IB. When an **`iopen`** is performed, an instrument driver consisting of all the devices at logical addresses contiguous to the base logical address is searched for. If no instrument driver will support the list of contiguous logical addresses, the device with the highest logical address will be removed and the search process repeated. This will continue until the driver is found or this list is exhausted. If no instrument driver is found the **`iopen`** call will fail.

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

In order to use the `iscpi` interface you must have installed the `SICL-VXI-ISCPI` fileset during the HP I/O Libraries installation. Additionally, you must have configured the system to include `iscpi` as an interface. See the *HP I/O Libraries Installation and Configuration Guide for HP-UX* for details on the installation and configuration.

Register-Based Instrument Drivers. `iscpi` includes drivers for most Hewlett-Packard register-based devices. These drivers are located in the directory specified during the `iscpi` interface configuration (default is **`/opt/sicl/lib/iscpi`**). Additionally, you can see the **`/opt/sicl/lib/iscpi/README`**`.iscpi` file for a list of currently supported register-based devices.

iscpi Device Session Example

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

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

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

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

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

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

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

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

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

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

}
```

Programming Directly to the Registers

When communicating with register-based devices, you 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 C-Size Command Module, HP B-Size Cardcage (built-in command module), or HP 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. The E1489 MXIbus EISA Interface (used to connect Series 700 to E1482) has 11 Mbyte maximum limit.

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.

If you are using an E1489 MXIbus Controller Interface, you can get 32-bit data reads and writes to VXIbus devices with D32 capabilities. Use the following table to determine which *map-space* argument to use with your SICL **imap/iunmap** function.

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

However, all accesses through these map windows can *only* be 32-bit transfers. The application software must do a 32-bit assignment to generate the access, and only accesses on 32-bit boundaries are allowed. If 8- or 16-bit accesses to the device are also necessary, a normal **I_MAP_A16/24/32** map must also be requested. These restrictions are specific to the E1489 interface.

I_MAP_EXTEND is not supported for 32-bit access with the E1489. If you need 32-bit access with the E1489, use one of the parameters listed above with _D32 attached.

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 the *HP SICL Reference Manual* 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 and 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 and 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 buss 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);
}
```

Enabling V743 Shared Memory

You can reserve 1 Mbyte of the V743's system memory to be used as VXI shared memory in A24 address space. The V743 is shipped with this feature disabled. When enabled, 1 Mbyte of the system memory becomes unavailable for use by the operating system and your applications. Instead, this memory is mapped onto the VXI backplane for use as VXI shared memory. You can use the SICL `e1497cnf` utility to enable or disable this feature. The following is an example of running the utility on the `vxi` interface:

```
e1497cnf -i vxi
```

When run, the above utility prompts asking if it is OK to reboot the system. If shared memory is currently disabled, then the utility will enable it. If shared memory is currently enabled, then the utility will disable it. See Appendix E, "Customizing Your VXI/MXI System," for more information on using the `e1497cnf` utility.

NOTE

You must be superuser to run this utility. This utility also requires you to reboot the system.

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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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`**, and so forth.

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

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

NOTE

The only interface session operations supported by I-SCPI are service requests and locking.

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** file (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.)

HP 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 such as an HP E1497/98 V743 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 HP controllers:

V743 VME Mapping Support

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

E1489 VME Mapping Support

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

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 the *HP SICL Reference Manual* 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* for information on configuring for `VME Only`. Also see `isetintr` in the *HP SICL Reference Manual* 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 card cage. The card cage 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 V743 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 HP 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.

Interpreted SCPI (`iscpi`) Device Sessions

The `iscpi` interface is used to program VXI register-based instruments. However, the VXI specific and register-based specific SICL functions, such as `ivxiws`, `imap`, and `ipeek` are not necessary, and therefore, are not implemented for the `iscpi` interface.

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

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

Interpreted SCPI (`iscpi`) Device Session Interrupts. The `iscpi` interface does not support interrupts. Therefore, the SICL `ionintr` function is not implemented for `iscpi` device sessions. There are no device-specific interrupts for the `iscpi` interface.

Interpreted SCPI (`iscpi`) Device Session Service Request. `iscpi` device sessions support Service Requests (SRQ) in the same manner as HP-IB. When one device issues an SRQ, *all* `iscpi` device sessions that have SRQ handlers installed (see `ionsrq` in the *HP SICL Reference Manual*) will be informed. This is an emulation of how HP-IB handles the SRQ line. The interface cannot distinguish which device requested service, therefore, `iscpi` acts as if all devices require service. Your SRQ handler can retrieve the device's **status byte** by using the `ireadstb` function. The status byte can be used to determine if the instrument needs service. It is good practice to ensure that a device isn't requesting service before leaving the SRQ handler. The easiest technique for this is to service all devices from one handler.

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 (unless you're using the **iscpi** interface, see "Programming with Interpreted SCPI").

- *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, and so forth.)

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. If the `iscpi` interface is being used, 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. All servant devices will cease to function until the VXI resource manager runs and normal operation is re-established.

NOTE

I-SCPI interface sessions only support service requests and locking (`ionsrq`, `ilock`, and `iunlock`).

Using HP SICL Trigger Lines

The following table shows the relationship between SICL and Hewlett-Packard controllers for the trigger lines and BNC connectors. These values may be passed to the **ivxitrig** or **isetintr** function:

NOTE

The E1489 is Hewlett Packard's MXIbus EISA Interface Card.

Trigger Lines

SICL	V743	E1489
I_TRIG_TTL0	TTL0	TTL0
I_TRIG_TTL1	TTL1	TTL1
I_TRIG_TTL2	TTL2	TTL2
I_TRIG_TTL3	TTL3	TTL3
I_TRIG_TTL4	TTL4	TTL4
I_TRIG_TTL5	TTL5	TTL5
I_TRIG_TTL6	TTL6	TTL6
I_TRIG_TTL7	TTL7	TTL7

Table continued on the next page.

Trigger Lines (cont.)

SICL	V743	E1489
I_TRIG_ECL0	ECL0	INVALID
I_TRIG_ECL1	ECL1	INVALID
I_TRIG_ECL2	INVALID	INVALID
I_TRIG_ECL3	INVALID	INVALID
I_TRIG_EXT0	Trig IN	INVALID
I_TRIG_EXT1	Trig OUT	INVALID
I_TRIG_EXT2	INVALID	INVALID
I_TRIG_EXT3	INVALID	INVALID
I_TRIG_CLK0	16 MHz Clock*	INVALID
I_TRIG_CLK1	INVALID	INVALID
I_TRIG_CLK2	INVALID	INVALID
I_TRIG_CLK10	INVALID	INVALID
I_TRIG_CLK100	INVALID	INVALID

* 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);
}
```

Routing VXI TTL Trigger Lines in a VXI/MXI System

When you have multiple card cages connected via the MXIbus, the TTL trigger lines are not routed from one card cage to another. The INTXbus does not allow multiple INTXbus devices to drive the same TTL trigger line. If you need TTL trigger lines in the extended VXI card cages, you need to edit the `ttltrig.cf` configuration file to map the TTL trigger line to the source logical address. See Appendix E, "Customizing Your VXI/MXI System," for information on editing this file.

The following example illustrates an entry in the `ttltrig.cf` file:

```
0 0
1 0
2 0
3 0
4 0
5 0
6 0
7 0
```

(Multiple trigger sources are still allowed on the same line within the same card cage.)

Where the first column is the TTL trigger line and the second column is the logical address of the TTL trigger source. Therefore, in the example above, all TTL trigger lines are sourced by the device at logical address 0. The following is an example of what you would see when the VXI resource manager runs:

```
VXI-MXI TTL Trigger Routing:

Name          0 1 2 3 4 5 6 7
----          - - - - - - - -
hpvximxi      I I I I I I I I
 I - MXI->VXI
 O - VXI->MXI
 * - Not Routed
```

Now the following illustrates TTL trigger line 1 being sourced by the device at logical address 24:

`ttltrig.cf` file:

```
0 0
1 24
2 0
3 0
4 0
5 0
6 0
7 0
```

Resource Manager output:

```
VXI-MXI TTL Trigger Routing:

Name           0 1 2 3 4 5 6 7
----           - - - - - - - -
hpvximxi       I O I I I I I I
 I - MXI->VXI
 O - VXI->MXI
 * - Not Routed
```

NOTE

You can use the `e1489trg` diagnostic test to test the MXI/INTX trigger and interrupt circuitry. See Appendix E, "Customizing Your VXI/MXI System," for information on this and other diagnostic tests for the E1489 MXIbus Controller Interface.

Routing External Trigger Lines on the E1482 VXI-MXI Extender Bus Card

In order to use external triggers while using the HP E1482 VXI-MXI Bus Extender card, you must route the external trigger lines to the TTL trigger lines. This can be done by using the `oride.cf` configuration file. This file contains values to be written to logical address space for register-based instruments. This data is written to the address space after the VXI resource manager runs, but before the system's resources are released. See Appendix E, "Customizing Your VXI/MXI System," for information on editing this file.

The following illustrates an entry in the `oride.cf` configuration file to route `Trig In` to TTL trig 1 and `Trig Out` to TTL trig 0:

```
1 2E 0x0302
```

Where 1 is the logical address of the VXI-MXI Bus Extender card, 2E is the offset value that corresponds to the MXIbus Trigger Configuration Register, 0x0302 is the value written to the register that will route `Trig In` to TTL trig 1 and `Trig Out` to TTL trig 0:

MXIbus Trigger Configuration Register

Bits 15 - 8	Bits 7 - 0
0 0 0 0 0 0 1 1	0 0 0 0 0 0 1 0

Bits 15 - 8 enable the corresponding VXIbus TTL trigger lines (TTL trig 7 - 0 respectively). And in the above table, TTL trigger lines 0 and 1 are enabled. Bits 7 - 0 determine the direction in which the corresponding TTL trigger lines are mapped to the front panel SMB connectors. If both bits are set, then the corresponding trigger line is driven by `trig in`. If the TTL trigger line is enabled (TTL trig 15 - 8), and the corresponding bit (bits 7 - 0) is not set, then the corresponding trigger line is driven by ''trig out''.

See the *HP E1482 VXI-MXI Bus Extender User's Manual* for more information about writing to the MXIbus Trigger Configuration Register.

NOTE

Once you route the external trigger lines to use the TTL trigger lines, you must also edit your program to trigger from the TTL trigger lines instead of the external trigger lines.

Inverting the Polarity of the V743 External Trigger Lines

At times you may wish to change the polarity of the Trig In and Trig Out lines on the V743 VXI Controller. This would allow you to connect to an external device independent of the device's polarity. There is a SICL utility, `itrginvrt`, that can be used to do just this. The following is an example that inverts the polarity of the Trig In line:

```
itrginvrt -a vxi -i ON -o OFF
```

Where:

- `-a vxi` specifies the interface name, `vxi`
- `-i ON` specifies that the Trig In line is to be inverted
- `-o OFF` specifies that the Trig Out line is not to be inverted

See Appendix E, "Customizing Your VXI/MXI System," for a complete description of the `itrginvrt` utility.

NOTE

The external trigger lines remain inverted until power is cycled or the VXI resource manager runs (with `iclear` or `ivxirm`, for example). The external trigger lines then return to the same state as the trigger line routed to them.

Using i?blockcopy for DMA Transfers with the V743

The V743 VXI Controller has 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. Note that **I_MAP_SHARED** is not supported for DMA transfers.
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 the *HP SICL Reference Manual* 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 HP SICL with RS-232

Using HP SICL with RS-232

RS-232 is a serial interface that is widely used for instrumentation. Although it is slow in comparison to HP-IB or VXI, its low cost makes it an attractive solution in many situations. Because SICL for HP-UX 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 HP I/O Libraries installation. See the *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 HP-IB 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 HP-IB.

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

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

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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 "HP 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`, and so forth.

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");
```

HP SICL Function Support with RS-232 Device Sessions

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

`iprintf`, `iscanf`, `ipromptf`	SICL's formatted I/O routines depend on the concept of an EOI indicator. Since RS-232 does not define an EOI indicator, SICL uses the newline character (`\n`) by default. You cannot change this with a device session; however, you can use the `iserialctrl` function with an interface session. See the section titled "HP SICL Function Support with RS-232 Interface Sessions" later in this chapter.
`ireadstb`	Sends the IEEE 488.2 command `"*STB?"` to the instrument, followed by the newline character (`\n`). It then reads the ASCII response string and converts it to an 8-bit integer. Note that this will work only if the instrument 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 "HP SICL Function Support for RS-232 Interface Sessions" later in this chapter.

RS-232 Device Session Interrupts

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

RS-232 Device Session 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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`, and so forth.

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

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

HP SICL Function Support with RS-232 Interface Sessions

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

`iwrite`, `iread`	All I/O functions (non-formatted and formatted) work the same as for device sessions. However, it is recommended that all I/O be performed with device sessions to make your programs easier to maintain.
`ixtrig`	Provides a method of triggering using either the DTR or RTS modem 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 HP-IB. On an HP-IB interface, a device can request service from the controller by asserting a line on the interface bus. RS-232 does not have a specific line assigned as a service request line. 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 the *HP SICL Reference Manual* 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 HP SICL with LAN

Using HP SICL with LAN

This chapter explains how to use SICL over LAN (Local Area Network). LAN is a natural way to extend the control of instrumentation beyond the limits of typical instrument interfaces. 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 HP 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

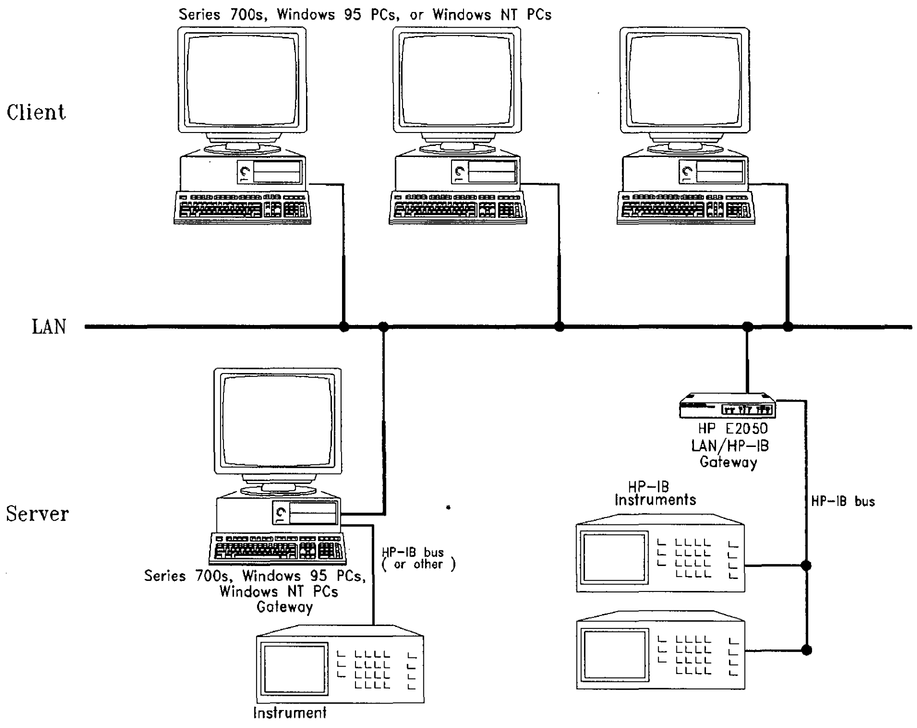
Overview of HP SICL LAN

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 Microsoft® Windows 95® or Windows NT® PC, or an HP E2050 LAN/HP-IB Gateway). The LAN server is connected to the instrumentation or devices that must be controlled. Once the LAN server has completed the requested operation on the instrument or device, the LAN server sends a reply to the LAN client. This reply contains any requested data and status information which indicates whether the operation was successful.

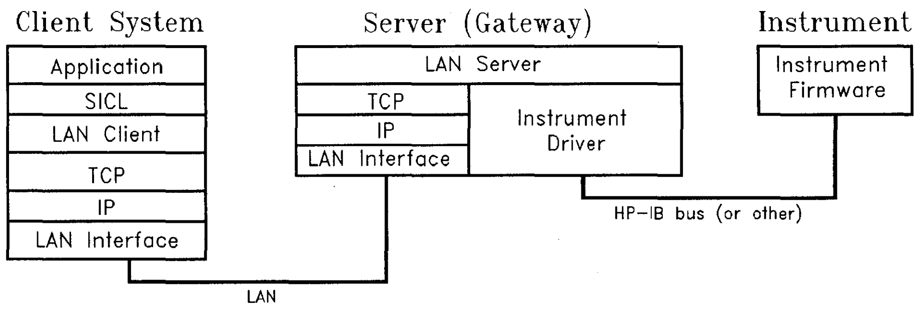

Series 700s, Windows 95 PCs, or Windows NT PCs
Client
LAN
Server
Series 700s, Windows 95 PCs, Windows NT PCs Gateway
HP-IB bus (or other)
Instrument
HP E2050 LAN/HP-IB Gateway
HP-IB Instruments
HP-IB bus

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 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 HP-UX currently supports SICL operations over the LAN to HP-IB/GPIB and VXI interfaces. Also, some SICL operations are not supported when using the TCP/IP Instrument Protocol. See the section titled "HP SICL Function Support with LAN-gatewayed Sessions" later in this chapter.

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

You can use both LAN networking protocols with a LAN client. To do so, simply configure *both* the SICL LAN Protocol and the TCP/IP Instrument Protocol on the LAN client system via the **iosetup** utility. (See the *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 section "Communicating with Devices Over LAN" 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.

HP SICL LAN Server

SICL includes the necessary software to allow a Series 700 workstation 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 must have a local interface configured for I/O. The supported interfaces for this release are HP-IB, VXI/MXI, and RS-232 for the SICL LAN protocol and GPIB/HP-IB and VXI interfaces for the TCP/IP Instrument Protocol. See the "HP 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 HP representative for a current list of other HP 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 "HP SICL Function Support with LAN-gatewayed Sessions" section later in this chapter). This means most operations you might request of an interface, such as HP-IB, connected directly to your controller, you can 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 system, 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 system 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 system 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 system with IP address `128.10.0.3`.
`lan[intserv]:GPIB`	An interface address corresponding to the GPIB interface attached to the system 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 system named **`intserv`** (30 is the default logical unit for LAN).
`lan[intserv]:GPIB,cmdr`	A commander session with the GPIB interface attached to the system named **`intserv`** (assumes that the server supports GPIB commander sessions).

NOTE

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

incorrect

```
lan,128.10.0.3:hpib
```

correct

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

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

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

HP SICL Function Support with LAN-gatewayed Sessions

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

The following 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 HP-IB specific functions
- All VXI specific functions
- All Serial specific functions
- `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 HP-IB device session via a LAN-to-HPIB gateway. Note that this example is the same as the first example in the "Using HP SICL with HP-IB" chapter, only the addresses passed to the **`iopen`** calls are modified. The example addresses assume the system with hostname **`instserv`** is acting as a LAN-to-HPIB 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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)

HP SICL Function Support with LAN Interface Sessions

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

- All HP-IB specific functions
- All 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`.
`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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 the *HP SICL Reference Manual* for details of the **ilantimeout** and **ilangettimeout** functions.

Note that a timeout value of `1` used with the `ilantimeout` function has special significance, causing the LAN client to not wait for a response from the LAN server. However, the timeout value of `1` should be used in special circumstances only and should be used with extreme caution. For more information about this timeout value, see the section, "Using the No-Wait Value," under the `ilantimeout` function in the *HP SICL Reference Manual.*

Default LAN Timeout Values

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

See the *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX*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 LAN server on HP-UX, 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. An HP-UX 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 HP SICL Program," for more details. Also see the documentation of the server you are using for the method to be used to reset the server.

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 HP-UX 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 HP SICL Program

Troubleshooting Your HP 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 HP SICL
- Troubleshooting HP SICL over LAN
- Troubleshooting HP SICL over RS-232
- Troubleshooting HP 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 the *HP SICL Reference Manual* 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 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 via the I/O Setup utility (**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, and so forth 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 (for example, **"hpib,57"**).
I_ERR_IO	Generic I/O error	An I/O error has occurred for this communication session.
I_ERR_LOCKED	Locked by another user	Resource is locked by another session (see **isetlockwait**).

Error Codes and Messages (continued)

Error Code	Error String	Description
I_ERR_NOCMDR	Commander session is not active or available	Tried to specify a commander session when it is not active, available, or does not exist.
I_ERR_NOCONN	No connection	Communication session has never been established, or connection to remote has been dropped.
I_ERR_NODEV	Device is not active or available	Tried to specify a device session when it is not active, available, or does not exist.
I_ERR_NOERROR	No Error	No SICL error returned, function return value is zero (0).
I_ERR_NOINTF	Interface is not active	Tried to specify an interface session when it is not active or available, or does not exist.
I_ERR_NOLOCK	Interface not locked	An **iunlock** was specified when device was not locked.
I_ERR_NOPERM	Permission denied	Access rights violated.
I_ERR_NORSRC	Out of resources	No more system resources available.
I_ERR_NOTIMPL	Operation not implemented	Call not supported on this implementation. The request is valid, but not supported on this implementation.
I_ERR_NOTSUPP	Operation not supported	Operation not supported on this implementation.
I_ERR_OS	Generic O.S. error	SICL encountered an operating system error.
I_ERR_OVERFLOW	Arithmetic overflow	Arithmetic overflow. The space allocated for data may be smaller than the data read.
I_ERR_PARAM	Invalid parameter	The constant or parameter passed is not valid for this call.
I_ERR_SYMNAME	Invalid symbolic name	Symbolic name passed to **iopen** not recognized.
I_ERR_SYNTAX	Syntax error	Syntax error occurred parsing address passed to **iopen**. Make sure that you have formatted the string properly. White space is not allowed.
I_ERR_TIMEOUT	Timeout 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 HP 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. Also, if you need to link with the archive library (HP-UX 9 only), make sure to include the **`-Wl,-E`** and **`-ldld`** compile/link options. See "Compiling and Linking an HP 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 HP 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 systems. 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). 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 error

`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 system. 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 system 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, and so forth).

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 the "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` (in `/opt/sicl/bin` directory) as `root`? If not, do so. If so, 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
rpc stream tcp nowait root /opt/sicl/bin/siclland 395183 1
siclland -l /var/opt/sicl/siclland_log
```

(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`:

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

`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 HP I/O Libraries configuration utility. See the *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 `sicllanḍ` server process. You can determine what `sicllanḍ` server processes are running by typing the following:

```
ps -ef | grep sicllanḍ
```

You will see something like the following:

```
root  2492  2480 11 15:33:27 ?      0:00 sicllanḍ -l /var/opt/sicl/sicllanḍ_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 `sicllanḍ`, 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/sicllanḍ_log`. See `sicllanḍ(1m)` for details.
- If no logging has been enabled, then the server as a whole will need to be reset by killing all `sicllanḍ` 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`:

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

`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 HP SICL Over RS-232

Unlike HP-IB, special care must be taken to ensure that RS-232 devices are correctly connected to your computer. Verifying your configuration first can save many wasted hours of debugging time. 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX*, as well as to the *RS-232 Cables Addendum* included in your HP I/O Libraries product package for more information on correct cabling.

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

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 HP-UX **`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 HP 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 HP 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 E2074 cabling and handshake behavior. In either case, it is important to read the *HP 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 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:

```
/sbin/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 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 *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 HP SICL," for SICL compile/link instructions.
- Chapter 3, "Using HP 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 HP SICL," for a description of SICL features.
- The interface specific chapter for a description of valid addressing.
- *HP I/O Libraries Installation and Configuration Guide for HP-UX* for a description of the HP 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 HP SICL with LAN," for a description of LAN addressing and timeouts.
- Your network administrator.
- The *Installing and Administering LAN/9000 Software* manual.

A

The HP SICL Files

The HP SICL Files

This appendix list the files and directories created on your system for SICL on HP-UX version 10.20.

SICL-RUN Fileset

Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-RUN`	Files for customization.
`/opt/sicl`	The main SICL software directory.
`/opt/sicl/bin`	The SICL configuration tools, programs, and so forth.
`/opt/sicl/defaults`	Default versions of the **`hwconfig.cf`** and **`iproc.cf`** files.
`/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`	Files for the shared library.

SICL-PRG Fileset

Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-PRG`	Files for customization.
`/opt/sicl/lib`	The SICL library (libsicl.a).
`/opt/sicl/include`	The SICL header file (sicl.h).
`/opt/sicl`	The DIL to SICL migration document.
`/opt/sicl/bin`	The **`dil2sicl`** migration tool.
`/opt/sicl/share/examples`	The SICL example programs.

SICL-VXI Fileset

Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-VXI`	Files for customization.
`/sbin/lib/eisa`	The EISA configuration files (700's only).
`/opt/sicl/bin`	The VXI specific configuration files, including the resource manager program, **`ivxirm`**.
`/opt/sicl/defaults`	The default versions of the VXI configuration files.
`/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-VXI-ISCPI Fileset

Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-VXI-ISCPI`	Files for customization.
`/opt/sicl/lib`	The I-SCPI driver shared libraries. I-SCPI interrupts SCPI commands for VXI register-based instruments.
`/opt/sicl/defaults`	The default version of the **`hwconfig.cf`** file.
`/opt/sicl/lib/iscpi`	The VXI register-based instrument driver shared libraries. The **`README.iscpi`** file lists what register-based instruments are supported with I-SCPI.
`/opt/sicl/bin`	The I-SCPI server program (to be used by I-SCPI).

SICL-HPIB Fileset

Directories	Description of Files
/var/adm/sw/products/ SICL/SICL-HPIB	Files for customization.
/sbin/lib/eisa	The EISA configuration files (700's only).
/opt/sicl/defaults	The default versions of the HPIB configuration files.
/opt/sicl/lib	The HPIB driver binary modules, which are linked and inserted in the kernel by the SICL configuration programs. Also adds the driver shared libraries.

SICL-GPIO Fileset

Directories	Description of Files
/var/adm/sw/products/ SICL/SICL-GPIO	Files for customization.
/sbin/lib/eisa	The EISA configuration files (700's only).
/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 Fileset

Directories	Description of Files
/var/adm/sw/products/ SICL/SICL-RS232	Files for customization.
/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 Fileset

Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-LAN`	Files for customization.
`/opt/sicl/defaults`	The default versions of the LAN configuration files.
`/opt/sicl/lib`	The LAN driver shared libraries.

SICL-LANSVR Fileset

Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-LANSVR`	Files for customization.
`/opt/sicl/bin`	The SICL LAN server daemon and a LAN configuration utility.

SICL-MAN Fileset

Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-MAN`	Files containing copyright information.
`/opt/sicl/share/man1`	Files containing `man` pages for SICL user utilities.
`/opt/sicl/share/man1m`	Files containing `man` pages for SICL system administrator utilities.
`/opt/sicl/share/man3`	Files containing `man` pages for SICL function calls.
`/opt/sicl/share/man4`	Files containing `man` pages for SICL configuration files.

SICL-MAN-HPIB Fileset

Directories	Description of Files
/var/adm/sw/products/ SICL/SICL-MAN-HPIB	Files for customization.
/opt/sicl/share/man3	Files containing man pages for SICL GPIB specific function calls.

SICL-MAN-GPIO Fileset

Directories	Description of Files
/var/adm/sw/products/ SICL/SICL-MAN-GPIO	Files for customization.
/opt/sicl/share/man3	Files containing man pages for SICL GPIO specific function calls.

SICL-MAN-VXI Fileset

Directories	Description of Files
/var/adm/sw/products/ SICL/SICL-MAN-VXI	Files containing copyright information.
/opt/sicl/share/man1m	Files containing man pages for SICL VXI/MXI specific system administrator utilities.
/opt/sicl/share/man3	Files containing man pages for SICL VXI/MXI specific function calls.
/opt/sicl/share/man4	Files containing man pages for SICL VXI/MXI specific configuration files.

SICL-MAN-RS232 Fileset

Directories	Description of Files
/var/adm/sw/products/ SICL/SICL-MAN-RS232	Files containing copyright information.
/opt/sicl/share/man3	Files containing man pages for SICL RS-232 specific function calls.

SICL-MAN-LAN Fileset

Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-MAN-LAN`	Files containing copyright information.
`/opt/sicl/share/man3`	Files containing **man** pages for SICL LAN specific function calls.

SICL-MAN-LANSVR Fileset

Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-MAN-LANSVR`	Files containing copyright information.
`/opt/sicl/share/man1m`	LAN specific **man** pages.

SICL-DIAG Fileset

Directories	Description of Files
`/var/adm/sw/products/SICL/SICL-DIAG`	Files containing copyright information.
`/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 10.

Building SICL Applications on HP-UX 10

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 application on HP-UX 9.x was linked with the SICL shared library, then no modification or re-compiles are necessary.
- If your SICL application on HP-UX 9.x 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 10 with the SICL shared library. This is the recommended method. See "Compiling and Linking an HP SICL Program" in Chapter 2 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, as follows.

    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 HP 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

Porting DIL to SICL

Porting DIL to SICL

This appendix provides information for translating your DIL (Device I/O Library) code to SICL. It will no longer be ported to new controller platforms, and today is not supported on HP 9000 Series 700 Computers or VXI in HP-UX 8.0 or above. One of the benefits of using SICL is that it is designed for instrument I/O. It has some high level features such as formatted I/O and error handling. SICL provides easier device addressing and better platform and device independence than DIL.

The DIL to SICL Translation Process

The following procedure describes the steps involved in translating your DIL code to SICL. Use the `migration.doc` and the "Summary of DIL to SICL Conversion" table in the next section to translate your DIL code. Additionally, there's a section on "Recommendations for Your DIL to SICL Translation" later in this appendix. The `migration.doc` file can be found in the `/opt/sicl` directory.

1. Run the DIL to SICL translation utility located in the `/opt/sicl/bin` directory:

   `dil2sicl [-r` *file_nm*`] [-v] [-?]` *prog_nm1* `[ [prog_nm2] . . .]`

   Where:

`-r`*file_nm*	Writes a report instead of marking the DIL source code.
`-v`	Runs in verbose mode.
`-?`	Prints a usage message.
prog_nm[n]	Specifies file name or names of source code to process.

   This utility reads the *prog_nm1* DIL source code and generates two outputs: a *prog_nm*`.x` file that contains the DIL original program with in-line translation hints and a report to `stdout` that summarizes the DIL commands to be translated.

2. Edit the *prog_nm*`.x` file to contain strictly SICL commands.

   Areas that need attention are marked by the `SICL_DIL` string. You can search for this string and either follow the suggestions given by the `dil2sicl` utility, or you can redesign your I/O code to use SICL more efficiently. See "Recommendations for Your DIL to SICL Translation" later in this appendix.

3. Compile your new program and fix errors.
4. Test your new program.

Summary of DIL to SICL Conversion

Summary of DIL to SICL Conversion

System Function	SICL Equivalent
close	iclose
creat	iopen
dup	No SICL equivalent needed
fcntl	No SICL equivalent needed
open	iopen
read	iread
write	iwrite

Summary of DIL to SICL Conversion

DIL	SICL Equivalent
#include <dvio.h>	#include <sicl.h>
gpio_eir_ctl	*
gpio_get_status	igpiostat
gpio_set_ctl	igpioctrl
hpib_abort	iclear
hpib_address_ctl	igpibbusaddr
hpib_atn_ctl	igpibatnctl
hpib_bus_status	igpibbusstatus
hpib_card_ppoll_resp	igpibpollconfig
hpib_eoi_ctl	iwrite
hpib_io	iopen, igpibsendcmd, iwrite, iread
hpib_parity_ctl	*
hpib_pass_ctl	igpibpassctl
hpib_ppoll	igpibppoll
hpib_ppoll_resp_ctl	igpibppollconfig
hpib_ren_ctl	igpibrenctl
hpib_rqst_srvce	igpibsetstb
hpib_send_cmnd	igpibsendcmd
hpib_spoll	ireadstb
hpib_status_wait	iwaithdlr, isetintr, ionintr
hpib_wait_on_ppoll	*

Summary of DIL to SICL Conversion Continued

DIL	SICL Equivalent
io_burst	*
io_dma_ctl	ihint
io_eol_ctl	itermchr
io_get_term_reason	iread
io_interrupt_ctl	isetintr
io_lock	ilock
io_on_interrupt	ionintr, isetintr
io_reset	iclear
io_speed_ctl	ihint
io_timeout_ctl	itimeout
io_unlock	iunlock
io_width_ctl	igpiosetwidth

Summary of DIL to SICL Conversion Continued

DIL	SICL Equivalent
vxi_abet_ctl	*
vxi_bnc_trigger_ctl	ivxitrigroute
vxi_bus_complete_status	*
vxi_bus_status	ivxibusstatus
vxi_end_ctl	iwrite
vxi_get_a16_addr	imap
vxi_get_device_info	ivxirminfo
vxi_get_laddr	*
vxi_get_manufacturer	ivxirminfo
vxi_get_model	ivxirminfo
vxi_get_servants	ivxiservants
vxi_get_version	I_SICL_REVISION
vxi_get_vme_device_info	*
vxi_map_a16	imap
vxi_map_a24	imap
vxi_map_a32	imap
vxi_map_device	imap
vxi_map_shared	imap
vxi_rcv_ws_response	ivxiws
vxi_resp_method	*
vxi_rqst_srvce	*
vxi_select_a32_page	imap
vxi_select_commander	*
vxi_select_servant	iopen

Summary of DIL to SICL Conversion Continued

DIL	SICL Equivalent
vxi_send_ws_cmnd	ivxiws
vxi_set_signal	*
vxi_spoll	ireadstb
vxi_status_wait	ivxiwaitnormop
vxi_trigger	ixtrig
vxi_trigger_off	ivxitrigoff
vxi_trigger_on	ivxitrigon
vxi_unmap_a16	iunmap
vxi_unmap_a24	iunmap
vxi_unmap_a32	iunmap
vxi_unmap_device	iunmap
vxi_unmap_shared	iunmap
vxi_ws_trigger	itrigger

* No support in SICL.

Recommendations for Your DIL to SICL Translation

Since DIL and SICL are very different I/O libraries just translating the DIL code to SICL does not guarantee an optional DIL to SICL translation. In most situations the best SICL design is not the same as the original DIL design. There are a few features in SICL that could make your applications more readable and maintainable. These features are device sessions, error handling, and formatted I/O. See Chapter 3, "Using HP SICL," for a complete description of these SICL features and keep the following in mind:

- Device sessions allow communication with a device without knowing the specifics of the interface's communication method. SICL allows different types of communication sessions: device sessions, interface sessions, and commander sessions. With interface sessions the user must do all the bus maintenance for the interface.
- Formatted I/O can reduce the amount of code in an application significantly because it does all the conversion for you. SICL provides the **`iprintf`**, **`iscanf`**, and **`ipromptf`** formatted I/O routines for instrument I/O. These routines are geared towards instruments, with printing flags for 488.2, NR1, NR2, and NR3 numeric types.
- By installing an error handler, the need to continually check return values is now longer necessary. This can reduce the amount of code significantly. SICL provides the `ionerror` error handler. You can install this 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.

D

The HP SICL Utilities

The HP 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 HP-IB device will result in the DCL command being sent to that device. Executing `iclear` on an HP-IB 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

$$\texttt{ipeek}\ [\texttt{-v}]\ [\texttt{-?}]\ \begin{bmatrix}\texttt{-b}\\ \texttt{-w}\\ \texttt{-l}\end{bmatrix}\ \textit{sym\_name map\_space offset}$$

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 the *HP SICL Reference Manual* 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

$$\texttt{ipoke}\ [\texttt{-v}]\,[\texttt{-?}]\begin{bmatrix}\texttt{-b}\\ \texttt{-w}\\ \texttt{-l}\end{bmatrix}\ \textit{sym\_name map\_space offset value}$$

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 the *HP SICL Reference Manual* 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
```

E

Customizing Your VXI/MXI System

Customizing Your VXI/MXI System

This appendix describes what files you would edit to customize your VXI/MXI system. Additionally, the VXI/MXI specific utilities are described. This appendix contains the following sections:

- Overview of VXI/MXI Configuration
- The VXI/MXI Resource Manager
- The VXI/MXI Configuration Files
- The VXI/MXI Configuration Utilities
- Multiple V743 Configuration

Overview of VXI/MXI Configuration

When HP SICL is installed and configured according to the procedures in the *HP I/O Libraries Installation and Configuration Guide for HP-UX*, 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 card cage 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 card cage initialization unless otherwise specified in the **`iproc.cf`** configuration file (default is to run at card cage 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. In order to use VXI/MXI, you must have loaded this fileset during the installation. See the *HP I/O Libraries Installation and Configuration Guide for HP-UX* 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 card cage 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 card cage 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 appendix 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 card cage. 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 HP SICL with VXI/MXI," for more information on setting up VME devices in your VXI card cage. 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 HP 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 HP SICL with VXI/MXI," for an example of using this file.

The iproc Utility (Initialization and SYSRESET)

On HP-UX 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 directories:

```
/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. For example, vxi16 is used for the E1489 MXI Extender Card and the V743 VXI Controller. Run the I/O setup configuration utility for information on the **`Logical Unit`** of your VXI/MXI interface. Also see "VXI/MXI Configuration Utilities" later in this appendix for information on using this utility.

VXI/MXI Configuration Utilities

The following SICL utilities are available to help you configure your VXI/MXI system:

- `e1489mir`
- `e1489trg`
- `e1489tsh`
- `e1497cnf`
- `iproc`
- `itrginvrt`
- `ivxirm`
- `ivxisc`

The utilities are located in the `/opt/sicl/bin` directory. Each of these utilities is described in detail in the following sections.

e1489mir

Syntax `e1489mir -i` *interface_name* [`-v`][`-x`][`-?`]

Description This is a diagnostic test for the E1489 MXIbus Controller Interface. This tests the MXI control logic on the E1489. You must be superuser to run this test, and you must reboot the computer after the test has run.

The parameter definitions follow.

`i` *interface_name*	Specifies the interface, such as vxi.
`v`	Turns on verbose mode.
`x`	Tells the program not to print warnings.
`?`	Prints the usage of the **`e1489mir`** program.

Example

```
e1489mir -i vxi
```

e1489trg

Syntax `e1489trg -a` *log_addr* `-i` *interface_name* [`-v`] [`-x`] [`-?`]

Description This is a diagnostic test for the E1489 MXIbus Controller Interface. This test the MXI/INTX trigger and interrupt circuitry. This test requires connection to the E1482 VXI/MXI Bus Extender.. You must be superuser to run this test, and you must reboot the computer after the test has run.

The parameter definitions follow.

a *log_addr*	Logical address of the VXI/MXI device.
i *interface_name*	Specifies the interface, such as vxi.
v	Turns on verbose mode.
x	Tells the program not to print warnings.
?	Prints the usage of the `e1489trg` program.

Example

```
e1489trg -a 1 -i vxi
```

e1489tsh

Syntax `e1489tsh -i` *interface_name* [`-l`][`-v`][`-x`][`-?`][`-s` *start 64k page*] [`-c` *number 64k pages*]

Description This is a diagnostic test for the E1489 MXIbus Controller Interface. This is a memory test, primarily for testing shared memory on the E1489. You must be superuser to run this test, and you must reboot the computer after the test has run.

The parameter definitions follow.

`i` *interface_name*	Specifies the interface, such as vxi.
`l`	Use `I_MAP_SHARED` (otherwise use `I_MAP_A24`).
`v`	Turns on verbose mode.
`x`	Tells the program not to print warnings.
`?`	Prints the usage of the **`e1489tsh`** program.
`s` *start 64k page*	Location to start 64k test.
`c` *number 64k pages*	Number of 64k pages.

Example

```
e1489tsh -i vxi
```

e1497cnf

Syntax `e1497cnf -i` *interface_name* [`-v`][`-x`][`-?`]

Description This is a configuration utility for the V743 VXI Controller. When you first run this utility you must make a selection to configure one of the following: shared memory, VME bus request level, or VME bus request mode. Selecting shared memory allows you to reserve 1 Mbyte of the V743's system memory to be used as VXI shared memory in A24 address space. By default, this mode is disabled, no shared memory. VME bus request level allows you to select the level (0, 1, 2, or 3) to be used. VME bus request mode allows you to select `ROR`, `RWD`, or `Fair RWD` as your bus request mode.

You must be superuser to run this utility, and you must reboot the VXI controller after the utility has run.

The parameter definitions follow.

`i` *interface_name*	Specifies the interface, such as vxi.
`v`	Turns on verbose mode.
`x`	Tells the program not to print warnings.
`?`	Prints the usage of the `e1497cnf` program.

Example

```
e1497cnf -i vxi
```

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 `/etc/opt/sicl` directory.

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.

itrginvrt

Syntax

```
itrginvrt -a interface_name -i [ON|OFF]-o [ON|OFF][-v][-x]
[-?]
```

Description This is a configuration utility to be used with the V743 VXI Controller. This utility allows you to change the polarity of the V743 Trig In and Trig Out lines. The external trigger lines will remain inverted until power is cycled or the resource manager runs. The external trigger lines will then return to the same state as the trigger line routed to them. The default state is inverted.

The parameter definitions follow.

a *interface_name*	Specifies the interface, such as vxi.
i	Sets the state of Trig In, `ON` inverts the polarity.
o	Sets the state of Trig Out, `ON` inverts the polarity.
v	Turns on verbose mode.
x	Tells the program not to print warnings.
?	Prints the usage of the **itrginvrt** program.

Example

```
itrginvrt -a vxi -i ON -o OFF
```

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 card cage 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 card cage or multi-card cage 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 is **`/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                 0  0  0  0  0     0  0  0  0  0  0
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                 0           0  0  0  0  0  0  0  0
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
```

Multiple V743 Configuration

This section describes how to configure your system for multiple V743 VXI Controllers in a single VXI mainframe. Before configuring SICL, you must install the hardware into the mainframe. Keep in mind that you must change the logical address of the non-slot 0 V743. You are only allowed one V743 as slot 0 controller. Therefore, any other V743s in the mainframe must have a logical address other than 0.

NOTE

The resource manager should not be run manually or automatically by a script on any non-slot 0 V743. Also note that SICL does not support commander sessions on the V743.

Once you have installed the V743s into the mainframe, use the following procedure to configure SICL:

1. Login as `root` on the system to be configured.
2. Run the I/O setup utility to add each V743 in your system. The following command will invoke the I/O setup utility:

   ```
   /opt/sicl/bin/iosetup
   ```

3. The available interfaces will be listed on the left side of the window. Click on the V743 you wish to configure and click on the (Configure) button. The I/O setup utility will display information about that interface. Edit the fields displayed to match your setup. Ensure that the logical address of each V743 added is correct. If you need more information, click on the (Help) button for detailed information about the field content. When done, click on the (OK) button.

   Repeat this step for each interface you wish to configure.

NOTE

You must ensure that all addresses and interrupt lines (IRQs) are unique and do not conflict with an address or IRQ line used by any other card in the system. The utility will warn you of any conflicts.

4. Once you have added all desired interfaces, click on the (Done) key from the main menu. The I/O Setup utility will edit the `hwconfig.cf` file and rebuild the kernel. Status messages will scroll by very quickly. To review the messages, see the `/var/opt/sicl/kernbld.log` file.
5. Edit the `/etc/opt/sicl/iproc.cf` file for each non-slot 0 V743. Comment out the following lines with the `#` character:
   - sysreset (for example, sysreset vxi2 ivxirm -I vxi2 -t 5&)
   - monitor
   - boot -p -I vxi
6. If you want the non-slot 0 V743s to receive IRQs, then you must edit the `/etc/opt/sicl/vxi`*LU*`/irq.cf` file for the V743 in slot 0. Edit this file to exclude IRQ lines used by the other V743's. The `irq.cf` file uses the following format:

   IRQ_line handler_logical_address interruptor_logical_address

   The following is an example of an `irq.cf` file in a system that has three V743s at logical addresses 0, 16, and 24. These V743's have been configured to use IRQ lines 1, 2, and 3 respectively.

   ```
   2  16  16  #exclude line for V743 at logical address 16
   3  24  24  #exclude line for V743 at logical address 24
   ```

   There is no need to edit the `irq.cf` configuration file for the non-slot 0 V743s since this file is only used by the resource manager and the resource manager is only run on the slot 0 device.
7. Run the SICL configuration program on the other V743s in the system:

   ```
   /opt/sicl/bin/siclconf
   ```

This program will configure your system, including building a new kernel and running the `eisa_config(1m)` program, if necessary. While the program runs, it prompts with the following.

- `OK to use....`

  You may use other file names if you have customized this file. Note, however, that many applications expect to find the current kernel description in the location specified.
- `Save backup of ...`

  This is the name for the kernel backup file. The program copies your old kernel file to this new file. If you use the default name, the file will be overwritten with the latest cnode in a cluster upgrade. Use a unique name if you wish to have a specific file for a particular cnode.
- `Would you like to see the log file?`

8. When the SICL configuration program finishes, it will ask if you want to reboot your system. Type `y` to reboot at this time. If you want to manually reboot, type `n` and manually reboot by typing the following:

   ```
   /sbin/reboot
   ```

NOTE

With some current kernel configurations, the SICL configuration will be unable to complete until after the system has rebooted. If this happens, you will be informed of an unsuccessful completion and you will be asked to reboot your system. After you reboot, login as `root` and type the following:

```
/opt/sicl/bin/siclconf -e
```

The `-e` option runs only `eisa_config`.

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 system.

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 (i.e. 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.

interrupts
Asynchronous events 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 an interface. A logical unit, in SICL, uniquely identifies an interface. Each interface on the controller must have a unique logical unit. The logical unit is specified during the system configuration.

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. 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. This name uniquely identifies the interface on this controller. If there is more than one interface on the controller, each interface must have a unique symbolic name. The symbolic name is specified during the system configuration.

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

HEWLETT®
PACKARD

E2091-90004