HP Standard Instrument Control Library

User's Guide for LynxOS

Notice

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

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

Warranty Information

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

Restricted Rights Legend

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

Printing History

Edition 1 — August 1997

Conventions Used in This Manual

This manual uses the following typographical conventions:

Getting Started	Italicized text is used for book titles and for emphasis.
Dialog Box	Bold text is used for the first instance of a word that is defined in the glossary.
`File`	Computer font represents text that you will see on the screen, including menu names, features, buttons, or text that you have to enter.
`dir` *filename*	In this context, the text in computer font represents an argument that you type exactly as shown, and the italicized text represents an argument that you must replace with an actual value.
`File ⇒ Open`	The "⇒" is used in a shorthand notation to show the location of features in the menu. For example, "`File ⇒ Open`" means to select the `File` menu and then select `Open`.
`Sml | Med | Lrg`	Choices in computer font, separated with bars (|), indicate that you should choose one of the options.
Press **Enter**	In this context, bold represents a key to press on the keyboard.
Press **Ctrl + O**	Represents a combination of keys on the keyboard that you should press at the same time.

Contents

B. Customizing Your VXI System

C. Configuring HP SICL

1

Introduction

Introduction

Welcome to the *HP Standard Instrument Control Library (SICL): User's Guide for LynxOS*. This manual describes how to configure, and use SICL on LynxOS.

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

- **Chapter 2 - 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 3 - Using HP SICL with HP-IB** describes communicating over the HP-IB interface. Example programs are also provided.
- **Chapter 4 - Using HP SICL with VXI** describes communicating over the VXIbus. Example programs are also provided.

This guide also contains the following appendices:

- **Appendix A - The HP SICL Utilities** describes the SICL utilities you can use to read and write to devices or interfaces from the command line.
- **Appendix B - Customizing Your VXI System** explains how you can customize your VXI system. VXI configuration utilities are documented as well.
- **Appendix C - Configuring HP SICL** explains how to edit the hardware configuration file and run the SICL configuration utility.

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. Applications written in C or C++ using this library can be ported at the source code level from one system to another without, or with very few, changes.

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

The HP E6237A Pentium Real-Time Controller with SICL on LynxOS supports the following interfaces:

- VXIbus (including multiple mainframe systems using the E1482B VXI-MXI extender)
- HP-IB

HP SICL Features

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

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

HP SICL User

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

2

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.

This chapter contains the following sections:

- Compiling and Linking HP SICL Programs
- 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*.

Compiling and Linking HP SICL Programs

You can create your SICL applications using ANSI C or C++ by following the instructions described in the LynxOS documentation. When compiling and linking a C program that uses SICL, use the `-lsicl` command line option to link in the appropriate library. The following example creates the executable file for the program called `idn`:

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

- The `-o` option creates an executable file called `idn`.
- The `-lsicl` option links in the SICL library (`libsicl.a`).
- The `-mthreads` option allows for multi-threaded execution. (SICL requires the `-mthreads` option.)

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`. Then, once the variable is declared, 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 or interface address, or the string `cmdr` for a commander session. See the sections that follow for details on creating the different types of communications sessions.

Your program may have several sessions open at the same time by creating multiple `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.

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 interface.

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 or VXI). This gives you full control of the activities on a given interface, but does make code less portable.

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.

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, your program is acting like a device on the interface (non-controller).

Note Commander sessions are *not* supported on VXI.

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. 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 compact, but not fast.

- Non-formatted I/O sends or receives raw data to or from a device, interface, or commander. With non-formatted I/O, no formatting or conversion of the data is performed. Thus, if formatted data is required, it must be done by the user.

See the following sections for a complete description and examples of using formatted I/O and non-formatted I/O.

Formatted I/O

The SICL formatted I/O mechanism is similar to the C `stdio` mechanism. SICL formatted I/O, however, is designed specifically for instrument communication and is optimized for IEEE 488.2 compatible instruments. The three main functions for formatted I/O are as follows:

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

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

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

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

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

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

See 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 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`	Left causes 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);
```

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

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

Output Conversion Characters	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.

Input Conversion Characters	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 (located in `/usr/sicl/examples`) illustrates using the formatted I/O functions. 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\n", &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 (located in `/usr/sicl/examples`) 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`. The interrupt will then be finished before `iwaithdlr` is called. In this case, `iwaithdlr` may have nothing to wait for, and will wait until its timeout period is reached, if any. 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 ();
...
```

Protecting I/O Calls Against Interrupts

In SICL, I/O calls like `iread` and `iprintf` are 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 system behavior such as the reaction to 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 (located in `/usr/sicl/examples`) 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>

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);

  id = iopen ("vxi");
  iintroff();
```

```
  /* 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 (located in `/usr/sicl/examples`) 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 (located in `/usr/sicl/examples`) 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.

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

The recommended way to use locking is per transaction. Per transaction means that you lock before you set up 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 (located in `/usr/sicl/examples`) shows 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);
}
```

3

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. This chapter describes in detail how to open a communications session and communicate with HP-IB devices, interfaces, or controllers.

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 set up 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.

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`, etc.

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 mainframe. 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 selected device clear (SDC) on the device corresponding to this session.

HP-IB Device Session Interrupts

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

HP-IB Device Sessions and Service Requests

HP-IB device sessions support Service Requests (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 (located in `/usr/sicl/examples`) 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.

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`, etc.

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 interface 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 isn't

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 (located in `/usr/sicl/examples`) 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 (located in `/usr/sicl/examples`) sets up two HP-IB instruments over an interface session and has the instruments communicate with each other.

The three 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];

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

   /* 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, 4);

   /* 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");
   iintron();
}
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.

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`, etc.

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 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.

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

4

Using HP SICL with VXI

Using HP SICL with VXI

This chapter explains how to use SICL to communicate over the VXIbus. This chapter contains the following sections:

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

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

Creating a Communications Session with VXI

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

To begin programming your VXI 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 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 one of the following methods to program register-based instruments:

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

HP Command Module - Use a Command Module to interpret the high-level SCPI commands. The `hpib` interface is used with a Command Module.

C-SCPI - Use C-SCPI to convert high-level SCPI commands to register peeks and pokes.

Programming with message-based and register-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 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.

The following are example addresses for VXI device sessions:

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

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

Note

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

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

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

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

Message-Based Device Session Example

The following example program (located in `/usr/sicl/examples`) 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 you can use to communicate with register-based devices:

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, however.
HP Command Module	When you use an HP Command Module to communicate with VXI 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.
C-SCPI	See the manual, *HP E6237A Compiled SCPI for LynxOS User's Guide*.

Addressing VXI 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.

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

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

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

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

Programming Directly to the Registers

When communicating with register-based devices, you must send a series of peeks and pokes directly to the device's registers. 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.

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 device registers (device session only, 64 bytes).
`I_MAP_EXTEND`	Maps in VXI device extended memory address space in A24 or A32 address space (device sessions only).
`I_MAP_SHARED`	Maps in VXI A24/A32 memory that is physically located on the computer (sometimes called local shared memory, interface sessions only).

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);

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

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

Note

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

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. It is good practice to use the `iunmap` function to unmap the memory space when it is no longer needed.

Register-Based Programming Example

The following example program (located in `/usr/sicl/examples`) 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.

```
/* vxidev.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];
  volatile 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, (char *)base_addr, I_MAP_VXIDEV, 0, 1);

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

Communicating with VXI 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 cannot be used on other interfaces, and therefore, becomes less portable.

Addressing VXI Interface Sessions

To create an interface session on your VXI 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.

The following is an example address for VXI interface sessions:

`vxi`	An interface symbolic name.

Note

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

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

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

VXI Interface Session Example

The following example program (located in `/usr/sicl/examples`) 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

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 is often not implemented or conflicts with design decisions made by VME card vendors, some of the resources required to interface with VME cards may not be available. Therefore, there are certain limitations and requirements when using VME in a VXI system. Note that VME is not an officially supported interface for SICL.

WARNING **Physical damage may result by plugging some VME cards into a VXI mainframe. Some VME devices make specific use of the P2 connector on the backplane. This may conflict with the VXI definitions of these pins and may cause physical damage to the VME card or VXI mainframe. Verify that your VME card is compatible with the VXI mainframe before inserting the card.**

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

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

Note These steps are *not* normally used with VXI devices.

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 `/usr/sicl/etc/vxi1/vmedev.cf` on your system 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 number, 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. In a multiple mainframe system, use address ranges for the VME devices in each mainframe that are just higher than those used by the VXI devices in the same mainframe. To determine what A24 or A32 address ranges are used by your VXI devices, run the Resource Manager (`ivxirm`) without the VME devices installed. This is done automatically when the mainframe is powered on. 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.) Power down the mainframe, add the VME devices to it, and then power on the mainframe again.

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 `/usr/sicl/etc/vxi`*`lu`*`/oride.cf` file on your LynxOS system 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 same is true for the A24 and A32 address space, which may also require an entry in the `oride.cf` file.

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 is an example:

```
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. 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.

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

Supported Access Modes

The following table lists VME access modes supported on HP controllers:

	A16			A24			A32		
	D08	D16	D32	D08	D16	D32	D08	D16	D32
Supervisory 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.

See the *HP SICL Reference Manual* for a complete description of the `i?peek` and `i?poke` functions.

Unmapping Memory Space

If you want to unmap memory space when it is no longer needed, make sure you use the SICL `iunmap` function. 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 on the registers and register locations.

See the *HP SICL Reference Manual* for a complete description of the `iunmap` function.

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 as `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 `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 `/usr/sicl/etc/vxi`*`lu`*`/oride.cf` file on your system to open an A16 address window. The same is true for the A24 and A32 address space, which may also require an entry in the `oride.cf` file. 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 `/usr/sicl/etc/vxi`*`lu`*`/vmedev.cf` file on your LynxOS system to reserve the appropriate address range. The following is an example entry for a VME device in slot 6 of a VXI mainframe. The mainframe 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 `siclconf` utility.

The following ANSI C example program (located in `/usr/sicl/examples`) opens a VXI interface session and sets up an interrupt handler. When the `I_INTR_VME_IRQ1` interrupt occurs, the function defined in the interrupt handler 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 controller to be the handler for the VME IRQ1.

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

#define ADDR "vxi"

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

void main ()
{
  unsigned short reg;
  volatile 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);

  /* 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);
```

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

  /* 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);
}
```

HP SICL Function Support with VXI

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

Device Sessions

Message-Based Device Sessions

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

`iwrite`	Sends the data to the (message-based) servant using the word-serial write protocol and the *Byte Available* word-serial command.
`iread`	Reads the data from the (message-based) servant using the word-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 *Clear* to the specified message-based device.
`ionsrq`	Can be used to catch SRQs from message-based devices.

Register-Based Device Sessions

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

- Non-formatted I/O:

```
iread
iwrite
itermchr
```

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

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

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

Interface Sessions

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

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

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:

SICL	HP VXI Controller
I_TRIG_TTL0	TTLTRG0*
I_TRIG_TTL1	TTLTRG1*
I_TRIG_TTL2	TTLTRG2*
I_TRIG_TTL3	TTLTRG3*
I_TRIG_TTL4	TTLTRG4*
I_TRIG_TTL5	TTLTRG5*
I_TRIG_TTL6	TTLTRG6*
I_TRIG_TTL7	TTLTRG7*
I_TRIG_ECL0	ECLTRG0
I_TRIG_ECL1	ECLTRG1
I_TRIG_ECL2	INVALID
I_TRIG_ECL3	INVALID
I_TRIG_EXT0	Trig IN
I_TRIG_EXT1	Trig OUT
I_TRIG_EXT2	INVALID
I_TRIG_EXT3	INVALID
I_TRIG_CLK0	INVALID
I_TRIG_CLK1	INVALID
I_TRIG_CLK2	INVALID
I_TRIG_CLK10	INVALID
I_TRIG_CLK100	INVALID

The `itrigger` function, when used on a VXI 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.

Routing VXI TTL Trigger Lines in a VXI/MXI System

When you have multiple mainframes connected via the MXIbus, the TTL trigger lines are not routed from one mainframe 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 mainframes, you need to edit the `ttltrig.cf` configuration file to map the TTL trigger line to the source logical address. See Appendix B, "Customizing Your VXI 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 mainframe.)

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      O O O O O O O O
 I - MXI->VXI
 O - VXI->MXI
 * - Not Routed
```

Now the following illustrates TTL trigger line 1 being sourced by the device at logical address 129 in a second VXI mainframe:

`ttltrig.cf` file:

```
0 0
1 129
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      O I O O O O O O
 I - MXI->VXI
 O - VXI->MXI
 * - Not Routed
```

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

In order to use the external trigger ports on 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 B, "Customizing Your VXI System," for information on editing this file.

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

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 TRG 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 TRG 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.

Using i?blockcopy for DMA Transfers

The 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 (located in `/usr/sicl/examples`) illustrates using i?blockcopy 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 <sicl.h>

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);

   /* NOTE:  Shared memory is not supported.
             Use an array declared in the program or use malloc
   */

   shr_buf = malloc (0x80000);
   a24_buf = imap (id, I_MAP_A24, 0x20, 0x8, 0);

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

```
   printf("A24 to memory (D16).\n\n");
   iwblockcopy (id,
                (unsigned short *)a24_buf,
                (unsigned short *)shr_buf,
                1,
                0
               );

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

   printf("A24 to 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

Note SICL only supports interrupts on VXI/VME cards using Release on Acknowledgment (ROAK). VXI/VME cards using Release on Register Access (RORA) are *not* supported.

See the `isetintr` function in the *HP SICL Reference Manual* for a list of VXI 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;
  volatile char *a16_ptr = 0;

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

  /* Cause relay card to interrupt: */
  *(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 Specific Functions

Note

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

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

A

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 SDC 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.

The `iclear` command, when used on a VXI 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 `Clear` command to the specified device.

The parameter definitions follow.

`t` *timeout*	Times out after timeout milliseconds.
`v`	Turns on verbose mode.
`?`	Prints the usage of the iclear program.

Example

```
iclear -t 1000 vxi
```

ipeek

Syntax `ipeek [-v] [-?] [-b|-w|-l]` *sym_name map_space offset*

Description `ipeek` is the SICL utility for examining memory locations on interfaces that support mapping. The `ipeek` utility will print the contents of the specified memory location in hexadecimal.

The *sym_name* is the SICL `symbolic name` of the interface. The interface must support mapping, such as VXI.

The *map_space* is the map area that you would like to examine. Currently the only interface supported is VXI. The valid map spaces are A16, A24, A32, VXIDEV, EXTEND, and SHARED. 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 `ipoke [-v] [-?] [-b|-w|-l]` *sym_name map_space offset value*

Description `ipoke` is the SICL utility for writing to memory locations on interfaces that support mapping. The `ipoke` utility will write the contents of the value parameter to the specified memory location.

The *sym_name* is the SICL `symbolic name` of the interface. The interface must support mapping, such as VXI.

The *map_space* is the map area that you would like to write to. Currently the only interface supported is VXI. The valid map spaces are A16, A24, A32, VXIDEV, EXTEND, and SHARED. 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
```

B

Customizing Your VXI System

Customizing Your VXI System

When SICL is installed and configured, certain SICL utilities and configuration files are copied onto your system. The VXI system is configured using two SICL utilities and the VXI 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 mainframe resources. The resource manager reads the VXI configuration files and polls the VXI devices to determine their resources and capabilities. This utility runs at mainframe initialization unless otherwise specified in the `iproc.cf` configuration file (default is to run at mainframe initialization and when SYSRESET is detected).
configuration files	These files specify some site-dependent configuration rules and any changes from the default.

The VXI Resource Manager (ivxirm)

The `ivxirm` utility is the resource manager which initializes and configures the VXI mainframe resources. The resource manager reads the VXI 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 directory on your system:

```
/usr/sicl/etc/vxilu/rsrcmgr.out
```

where *lu* is the logical unit of the VXI 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 mainframe 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.

Note

If you manually re-run the resource manager and get a `GENERIC I/O error`, you need to terminate the `iproc` daemon, and execute the following command:

```
/usr/sicl/bin/iclear vxi
```

Generally, there is no need to manually run the resource manager.

The VXI 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 configuration files specify some site-dependent configuration rules and any changes from the default. These files reside in the following directories on your system. Each file is explained in the following sections.

File Name	LynxOS Directory Location
`vximanuf.cf`	`/usr/sicl/etc`
`vximodel.cf`	`/usr/sicl/etc`
`dynamic.cf`	`/usr/sicl/etc/vxi1`
`vmedev.cf`	`/usr/sicl/etc/vxi1`
`irq.cf`	`/usr/sicl/etc/vxi1`
`cmdrsrvt.cf`	`/usr/sicl/etc/vxi1`
`names.cf`	`/usr/sicl/etc/vxi1`
`oride.cf`	`/usr/sicl/etc/vxi1`
`ttltrig.cf`	`/usr/sicl/etc/vxi1`

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 device 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 mainframe. 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 the same resources aren't allocated to more than one device. See "Communicating with VME Devices" in Chapter 4, "Using HP SICL with VXI," for more information on setting up VME devices in your VXI mainframe. 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 4, "Using HP SICL with VXI," 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/MXI systems. See "Routing VXI TTL Trigger Lines in a VXI System" in Chapter 4, "Using HP SICL with VXI," for an example of using this file.

The `iproc` Utility (Initialization and SYSRESET)

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 on your system:

```
/usr/sicl/etc
```

The following is an example of the `/usr/sicl/etc/iproc.cf` file:

```
#
# For E623x support, Sample shown using SICL symbolic name
# as 'vxi'
#
boot echo "SICL: Instrument I/O Initialization"

boot ivxirm -I vxi

# When a SYSRESET occurs, rerun the resource manager
# (delay 5 sec).  The resource manager MUST be run in
# the background (ie. last character should be a '&').

sysreset vxi ivxirm -t 5
```

Viewing the VXIbus System Configuration

You can use the SICL `ivxisc` utility to read the current system configuration and print a human readable display by running the following command at the prompt:

```
ivxisc
```

See "VXI Configuration Utilities" later in this appendix for information on using this utility.

VXI Configuration Utilities

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

- `iproc`
- `ivxirm`
- `ivxisc`

The utilities are located in the following directory on your system:

```
/usr/sicl/bin
```

Each of these utilities is described in detail in the sections that follow.

iproc

Description `iproc` is designed to run at system boot time from `/etc/rc` on your system. It performs various SICL system initialization functions. In addition, it is configurable by the system administrator to execute programs at boot time or on certain asynchronous events, such as VXI SYSRESET. This configuration is done by editing the file `iproc.cf`, which is read only when the `iproc` daemon begins execution. It consists of lines beginning with keywords which determine the actions of the `iproc` program. The `iproc.cf` file is located in the following directory on your system:

```
/usr/sicl/etc
```

The format of the configuration lines is as follows:

keyword action

or

keyword interface name action

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.

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 if `sysreset` is not used. This is useful during debugging activities.

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 `/usr/sicl/etc/vxi`*lu* on your system, 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` *n* Delay *n* seconds before starting. To support the VXI Standard, set the delay to five seconds to allow instruments to complete their self test. If you do not set this option, the default value is no delay.

`v` Print a verbose output of the resource manager's actions. This is useful for debugging the mainframe configuration.

`D` The next argument specifies the directory that contains the `ivxisc` program. This defaults to `/usr/sicl/bin` on your system.

`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 mainframe or multi-mainframe 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 configuration files specify the site-dependent configuration rule changes. See "The VXI 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 power on the mainframe.

Example

```
ivxirm -p
```

ivxisc

Syntax `ivxisc` [`-sdvfphmi`] [*directory*]

Description The ivxisc command reads the operating configuration file, `/usr/sicl/etc/vxi`*lu*`/rsrcmgr.out` on your LynxOS system (where *lu* is the logical unit of the VXI 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 on your system. (default: `/usr/sicl/etc/vxi`*lu*)

Example For the VXI interface at logical unit (*lu*) 0:

```
ivxisc  /usr/sicl/etc/vxi0
```

A sample output follows.

`ivxisc` Output example:

```
VXI Current Configuration:


VXI Bus: 0
  Device Logical Addresses:  0  2  24  56
Slots:             0  1  2  3  4  5  6  7  8  9 10 11 12
                  -- -- -- -- -- -- -- -- -- -- -- -- --
Empty                 O  O  O  O  O     O     O  O  O  O
Single Device      X                 X     X
Multiple Devices
VME
Failed


VXI Device Table:

Name     LADD Slot Bus Manufacturer    Model
----     ---- ---- --- --------------- -----
dev1        0    0   0 Hewlett-Packard E623x Pentium VXI Controller w/Slot 0
relaymux    2    ?   0 Hewlett-Packard E1345 16 ch. 3W relay mux
dev2       24    8   0 Hewlett-Packard E1413A/B/C 64 ch. 100 Khz Scanning A/D
dev3       56    6   0 Hewlett-Packard E1415A 64 ch. Closed Loop Controller

 ? - slot number unknown


VME Device Table:

Name          Bus  Slot  Space  Size
----          ---  ----  -----  ----
No VME cards configured.


Failed Devices:

Name          Bus  Slot  Manufacturer      Model
----          ---  ----  ------------      -----
No FAILED devices detected.
```

`ivxisc` Output example (cont.):

```
Protocol Support (Msg Based Devices):

Name          CMDR SIG MSTR INT FHS SMP RG EG ERR PI PH TRG I4 I LW ELW 1.3
----          ---- --- ---- --- --- --- -- -- --- -- -- --- -- - -- --- ---
dev1           X    X   X               X     X      X                  X


Commander/Servant Hierarchy;

    dev1
        relaymux
        dev2
        dev3


Memory Map:

A24                  Device Name
---                  -----------
0x400000 - 0x7fffff  dev1
0x200000 - 0x23ffff  dev2
0x240000 - 0x27ffff  dev3

A32                      Device Name
---                      -----------
No devices mapped into A32 space.


Interrupt Request Lines:

                Handler        Interrupter
Name          1 2 3 4 5 6 7    1 2 3 4 5 6 7
----          - - - - - - -    - - - - - - -
dev1          X X X X X X X
relaymux
dev2
dev3
```

C

Configuring HP SICL

Configuring HP SICL

This appendix explains how to configure SICL. It includes procedures to edit the `hwconfig.cf` file, which contains the configuration details for SICL interfaces, and how to run the `siclconf` utility for rebuilding the kernel.

Configuring HP SICL for VXI

HP SICL is preconfigured for VXI at the factory. If you need to reconfigure SICL, use the following procedure.

The VXI configuration is done by running the `siclconf` utility, as explained in this section. The `siclconf` utility rebuilds the kernel. The following steps explain how to run this utility to configure VXI and rebuild SICL into the kernel.

1. Log in as root on the Lynx system.
2. Edit the `/usr/sicl/etc/hwconfig.cf` file.
3. Run the `siclconf` utility using the following command to build a new kernel:

   ```
   /usr/sicl/bin/siclconf
   ```

4. To use the new kernel, reboot the Lynx system using the following command:

   ```
   /etc/reboot -aN
   ```

5. Additional information is located in the README file `/usr/sicl/lib/README`.

Editing the Hardware Configuration File

The hardware configuration file, `hwconfig.cf`, contains the configuration details for SICL interfaces, and is used by the `siclconf` utility for rebuilding the kernel.

To Edit the `hwconfig.cf` File

This configuration involves editing the `hwconfig.cf` file to specify your I/O interfaces, building the kernel, and rebooting the system.

1. Log in as `root` on the system to be configured.

2. Edit the `/usr/sicl/etc/hwconfig.cf` file to reflect the I/O hardware you want to use. You need to have one entry for each I/O interface in the system. The default `hwconfig.cf` file is located in the `/usr/sicl/defaults` directory.

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.

3. Run the SICL configuration utility by entering the following command at the prompt:

   ```
   /usr/sicl/bin/siclconf
   ```

   The `siclconf` utility will configure your system, and rebuild the kernel.

4. Once the SICL configuration utility finishes, you need to reboot the system. Enter the following command at the prompt:

   ```
   /etc/reboot -aN
   ```

About the Hardware Configuration File

Each line in the `hwconfig.cf` file corresponds to an interface card that will be used for instrument I/O. There is only one line for each interface card in the system. This file only needs to be edited if you are not running the I/O setup utility. You can view the default `hwconfig.cf` file in the directory `/usr/sicl/defaults`.

The format of each line is as follows:

lu symname cardname location [*card specific values*]

Where:

lu	Logical unit number of the card (0<*lu*<10000). Each interface card must have a unique logical unit number. The actual value used is not important, but you must remember this number in order to address the card in your application properly.
symname	A symbolic name for your card. Each card must have a unique symbolic name. This name may be used instead of the logical unit number to address an interface. The default symbolic name for your HP-IB interface should be `hpib`.
cardname	The specific name of the card. This is used to determine which driver to use.
location	The location of the card.

Card-specific information is described below for each possible card. In each case, the values specified are numbers and may be represented in either hexadecimal (using `0x...`), octal (using `0...`), binary (using `0b...`), or decimal (default).

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.

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

E

F

G

H

I

L

M

N

O

P

W