HEWLETT PACKARD

Using the HP-IB Interface and Command Library with DOS.

Instrument Tools for Windows.

Computer
Museum

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.

Copyright © Hewlett-Packard Company 1989, 1991

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

Microsoft®, MS-DOS®, and GW-BASIC® are U.S. registered trademarks of Microsoft Corporation.

Vectra® is a U.S. registered trademark of Hewlett-Packard Company.

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Printing History

First Edition - July 1989

Second Edition - December 1991

Contents

Computer Museum

1 Using the Command Library for HP-IB Control

This manual describes how to use the HP-IB Interface and Command Library for HP-IB control. Use this manual if you want to control instruments from a DOS-based language listed on the *Supported Languages* sheet.

This chapter shows how to set up and use the Command Library for HP-IB instrument control. The following chapters provide detailed information about how to use the Command Library for programming in various languages.

System Requirements

You must have the following components to set up and use the Command Library:

- An HP Vectra® PC or IBM PC/XT/AT (or compatible) computer with at least 256 kilobytes of memory and an MS-DOS® 3.1 or later operating system (or PC-DOS 3.1 or later).
- The HP 82335 HP-IB Interface, which should be installed. (See the *Installing the HP-IB Interface* booklet for instructions.)

  If you're using an earlier HP 82990A HP-IB Interface, see "Compatibility" later in this chapter.
- The HP-IB Tools disks, which contain software to provide HP-IB control commands for the programming language you intend to use.
- A supported programming language, such as Microsoft® QuickBASIC. This software is *not* part of the Command Library. (See the *Supported Languages* sheet for a list of languages.)

Programming Languages

The Command Library supports four categories of programming languages. The supported versions of each language are listed on the *Supported Languages* sheet.

- **GW-BASIC® and similar languages**, including HP Vectra BASIC, Microsoft GW-BASIC, and IBM BASICA.
- **QuickBASIC and similar languages**, including Microsoft QuickBASIC, Microsoft QBasic and Microsoft Compiled BASIC.
- **Pascal languages**, including Microsoft Pascal and Borland Turbo Pascal.
- **C languages**, including Microsoft C, Microsoft QuickC, and Borland Turbo C.

Each category is discussed in a separate chapter in this manual. See the appropriate chapter for specific programming information. General programming information follows below.

Overview of the Command Library

The HP-IB Command Library for MS-DOS is a series of commands that let you control HP-IB instruments with a personal computer.

The commands are available on the Library disks in several programming languages: BASIC (interpreted and compiled), Pascal, and C. See the *Supported Languages* sheet for a detailed list of languages. For convenience, example programs in many supported languages are also included on the disk. In addition, a READ.ME file gives you current notes about the Command Library.

The HP-IB interface card provides the necessary electrical and mechanical interface for IEEE-488 communication. See appendix B for more information about HP-IB operation.

HP-IB Control

The Library commands give you access to the IEEE-488 standard control lines and bus commands listed below.

	Command	Description
Control Lines:		
	IFC	Interface Clear
	ATN	Attention
	SRQ	Service Request
	REN	Remote Enable
	EOI	End Or Identify
Universal Bus Commands:		
	LLO	Local Lockout
	DCL	Device Clear
	SPE	Serial Poll Enable
	SPD	Serial Poll Disable
	PPU	Parallel Poll Unconfigure
Addressed Bus Commands:		
	GTL	Go To Local
	SDC	Selected Device Clear
	PPC	Parallel Poll Configure
	GET	Group Execute Trigger
	TCT	Take Control
Unaddress Bus Commands:		
	UNL	Unlisten
	UNT	Untalk

The HP-IB Interface and Command Library support the following HP-IB functions (indicated by their IEEE 488.1 capability codes): SH1, AH1, T5, TE5, L3, LE3, SR1, RL0, PP0, DC0, DT0, E2, C1, 2, 3, 4, 9.

Library Commands

The Library commands give you a great deal of programming flexibility. For example, you can enter or output data directly as strings, real numbers, or binary data.

The following commands perform over 80 percent of the I/O tasks in most applications. Because the Library has a built-in number builder and formatter, there's no need for the programmer to convert data between string and numeric format. This reduces the number of programming statements and the associated overhead. Alternatively, unformatted transfers maximize the data-transfer rate.

IOENTER	Enter a single real number from a device.
IOENTERA	Enter an array of real numbers from a device.
IOENTERAB	Enter IEEE-488.2 "arbitrary-block" data from a device.
IOENTERB	Enter an unformatted block of data from a device.
IOENTERF	Enter the contents of a file from a device or interface.
IOENTERS	Enter an ASCII string from a device.
IOOUTPUT	Output a single real number to a device.
IOOUTPUTA	Output an array of real numbers to a device.
IOOUTPUTAB	Output IEEE-488.2 "arbitrary-block" data to a device.
IOOUTPUTB	Output an unformatted block of data to a device.
IOOUTPUTF	Output the contents of a file to a device or interface.
IOOUTPUTS	Output an ASCII string to a device.

The following commands let you check instrument and bus status whenever your program requires it.

IOPEN	Set up a service-request interrupt (BASIC only).
IOPPOLL	Perform a parallel poll.
IOPPOLLC	Perform a parallel poll configure.
IOPPOLLU	Perform a parallel poll unconfigure.
IOSPOLL	Perform a serial poll.
IOSTATUS	Determine the status of the HP-IB interface.

The following commands let you transfer active control using the HP 82335 HP-IB card.

IOPASSCTL	Pass active control from the computer to a device on the bus.
IOREQUEST	Set up a serial poll status byte for the computer to request service from the active controller.
IOTAKECTL	Take active control from the currently active controller back to the computer.

The following commands give you access to various HP-IB control lines and bus commands.

IOABORT	Abort all interface activity.
IOCLEAR	Return a device to a known state.
IOCONTROL	Write information directly to the interface.
IODMA	Set up DMA control.
IOEOI	Control the interface EOI mode.
IOEOL	Define an end-of-line string for output.
IOFASTOUT	Enables high-speed output timing.
IOGETTERM	Determine the reason for a read termination.
IOLLOCKOUT	Disable device front panel operation.
IOLOCAL	Enable device front panel operation.
IOMATCH	Define a read termination character.
IOREMOTE	Place a device in REMOTE mode.
IORESET	Set the interface to its start-up configuration.
IOSEND	Send user-specified HP-IB commands.
IOTIMEOUT	Set a timeout value.
IOTRIGGER	Trigger a device.

Compatibility

This HP 82335 HP-IB Interface and Command Library differ from the earlier HP 82990A HP-IB Interface and Command Library.

Any program written and compiled using the *earlier* Command Library will run properly on the *current* HP-IB interface.

Any program written and compiled using the *current* Command Library will run properly on the *earlier* HP-IB interface—*except* that BASIC service-request interrupts using IOPEN and pass control functions are not supported by the earlier interface. All other current features of the Command Library are supported (see below).

The current Command Library differs from the earlier library in the following ways:

- The supported programming languages have changed.
- The current library and interface support service-request interrupts for BASIC languages. (This is not supported by the earlier interface.)
- The current library supports arbitrary-block and binary data transfers, which don't require formatting conversions.
- The current library supports file transfers.
- The current library supports high-speed and standard-speed data transfers for all output operations, whereas the earlier library used high-speed transfers for DMA output and standard-speed transfers for all other output.
- The current library does *not* support the obsolete Hewlett-Packard PC Instruments System, whereas the earlier library *did* support it. This means you can't control both HP-IB instruments and the PC Instruments System from the same program with the current library.
- The current library supports passing of active control.
- The current library does not support HP-IB disk and tape access.

Typical HP-IB Operations

The following sections of this chapter give general guidelines for typical HP-IB operations:

- Formatted-data transfers.
- Block-data transfers.
- File transfers.
- String transfers.
- Direct-memory-access (DMA) transfers.
- Service-request interrupts.
- Processing I/O errors.
- Addressing Library commands.

These guidelines apply to the *type* of operation, *not* to the programming language. Use this general information along with the specific information in the chapters that cover your programming language.

Data Transfers

The Command Library provides six types of data transfers, each with an input command and an output command. The type of transfer you use in a program depends in part upon these factors:

- The format of the data required by the HP-IB device.
- The type of program variable that holds or will hold the data.
- The transfer speed required by the program or application.

The following table shows some of the differences among the types of data transfers. It can help you decide which type of transfer to use for a particular application.

Type of Transfer	**Commands**	**HP-IB Device Format**	**Program Variable**	**Transfer Speed**
Formatted Number	IOENTER IOOUTPUT	ASCII string	Real number	Slower
Formatted Array	IOENTERA IOOUTPUTA	ASCII string	Real number	Slower
String	IOENTERS IOOUTPUTS	ASCII string	String (character array)	Faster
Arbitrary-Block Data (IEEE-488.2)	IOENTERAB IOOUTPUTAB	IEEE-488.2 (numeric or string data)*	Real number, integer, or string (character array)*	Faster
Binary Block Data	IOENTERB IOOUTPUTB	Numeric or string data*	Real number, integer, or string (character array)*	Faster
File	IOENTERF IOOUTPUTF	Numeric or string data	Real number, integer, or string (character array)	Slower
* String data not supported for BASIC. Use IOENTERS or IOOUTPUTS.				

Certain other capabilities of the six types of transfers will help you decide which type to use. Detailed information about these differences is given in the sections that follow.

Type of Transfer	**Commands**	**DMA Transfer Supported?**	**Byte-Swapping Supported?**
Formatted Number	IOENTER IOOUTPUT	No	Not applicable
Formatted Array	IOENTERA IOOUTPUTA	No	Not applicable
String	IOENTERS IOOUTPUTS	Yes	Not applicable
Arbitrary-Block Data (IEEE-488.2)	IOENTERAB IOOUTPUTAB	Yes*	Yes*
Binary Block Data	IOENTERB IOOUTPUTB	Yes*	Yes*
File	IOENTERF IOOUTPUTF	No	No
* DMA and byte-swapping not supported simultaneously.			

In the table above, "DMA transfer" refers to using the computer's ability to transfer data directly to and from memory—the data bypasses the computer's main processor. "Byte-swapping" refers to the computer's ability to adjust for data-transfer differences between an HP-IB device and the computer—many devices send and receive numerical data with the most-significant byte first, whereas the computer stores and retrieves numerical data with the least-significant byte first. These aspects of data transfer are discussed in more detail next.

Input transfers differ in the conditions that end the input. Output transfers differ in the actions that occur at the end of the output. The following table summarizes input and output termination.

Input Commands and Terminating Conditions	**Output Commands and Terminating Actions**
IOENTER EOI sensed true (if enabled) Linefeed received	IOOUTPUT EOL added EOI set true (if enabled)
IOENTERA EOI sensed true (if enabled) Linefeed received Maximum number of values received	IOOUTPUTA EOL added EOI set true (if enabled)
IOENTERAB Coded number of bytes received (definite) Linefeed with EOI true (indefinite) Maximum number of bytes received *	IOOUTPUTAB (nothing) *
IOENTERB EOI sensed true (if enabled) Match character with EOI true (if enabled) Maximum number of bytes received *	IOOUTPUTB EOL added EOI set true (if enabled) *
IOENTERF EOI sensed true (if enabled) Match character (if enabled) Maximum number of characters received	IOOUTPUTF EOL added EOI set true (if enabled)
IOENTERS EOI sensed true (if enabled) Match character (if enabled) Maximum number of characters received *	IOOUTPUTS EOL added EOI set true (if enabled) *
* For BASIC only, ended by high-priority SRQ during DMA (if enabled).	

If you want to increase the speed of data transfers, you have several options:

- Use block-data transfers. Arbitrary-block and binary transfers can be 10 times faster than formatted transfers. (Also, string transfers are much faster than formatted transfers.)
- Use the DMA option. DMA can speed transfers by 15 percent.
- Use high-speed bus timing (output only). High-speed timing can speed output transfers by 30 percent.

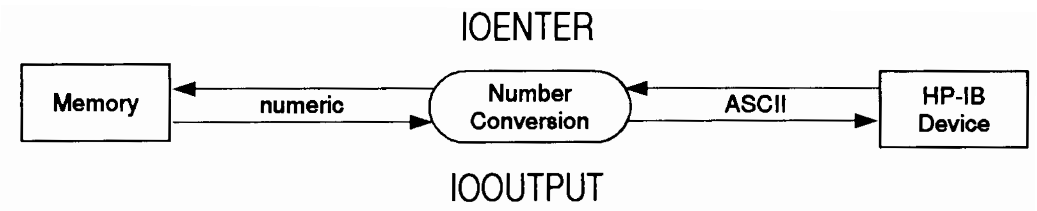
The following topics describe various types of data transfers.

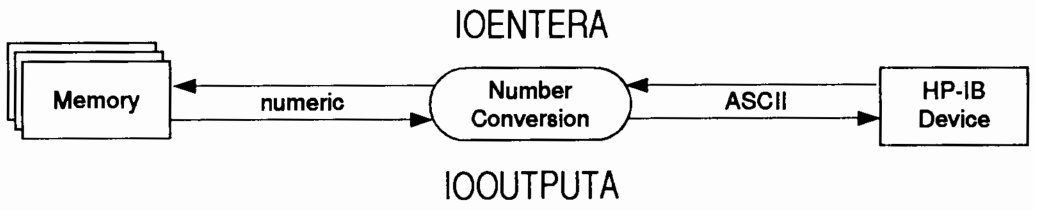
Formatted-Data Transfers

Many HP-IB devices send and receive numeric data as a string of ASCII characters—digits that represent the value. For example, the value 14.500 can be represented by the characters "1 4 . 5"—although this string of characters has no numeric significance to the computer. However, the string of characters is readily converted to and from a numeric value. Such "formatted data" provides a convenient method for transferring data.

Operation

A transfer of formatted data necessarily involves converting between ASCII characters and numeric values. The Command Library builds this conversion into the IOENTER, IOENTERA, IOOUTPUT, and IOOUTPUTA commands. If your HP-IB device sends and receives numeric data as ASCII characters, your program can use these commands to exchange data with the device. The following diagrams represent these transfers.





Formatted-data transfers require the use of the computer's processor to convert the data. Therefore, these transfers can't use direct-memory access (DMA). The automatic data-conversion process also takes a small amount of time, making formatted-data transfers slower than block-data transfers. However, formatted-data transfers are usually satisfactory for all but the most time-critical of transfers.

A valid number must contain only the allowable characters (`0123456789+-.Ee`) in a correct sequence—any other character indicates the end of the number. Numbers must be 0 or in the range $\pm 10^{-38}$ to $\pm 10^{38}$. For array input, nonnumeric characters separate the numbers. The following examples illustrate the conversion from characters to numbers:

`1.25`	*Enters the value 1.25.*
`-4.5E3`	*Enters the value* -4.5×10^{3}.
`.00055abc`	*Enters the value 0.00055.*
`1,234@567`	*Enters three values: 1, 234, and 567.*

Optional Conditions

Formatted-data transfers support the following options:

- Disabling EOI for input and output. The HP-IB End Or Identify line usually goes active for the last data byte transferred, but you can disable this convention—use the IOEOI command.
- Changing the EOL string for output. Data is usually terminated by carriage-return and linefeed characters, but you can change or eliminate these characters—use the IOEOL command.
- Setting the HP-IB transfer timing for output. Data is usually output with standard timing, but you can set high-speed timing—use the IOFASTOUT command.

Ending Formatted Input

You can use the IOENTER and IOENTERA commands to enter numeric data—one value or an array of values. The input operation ends when one of these conditions occurs:

- A data byte is received with the HP-IB EOI line active. You can prevent the EOI line from ending the transfer by using the IOEOI command to disable EOI.
- A linefeed character is received (after a valid number has been received for IOENTER, or after the maximum number of values you specified in IOENTERA is received).

Ending Formatted Output

You can use the IOOUTPUT and IOOUTPUTA commands to send numeric data—one value or an array of values. At the end of the output operation these events occur:

- The EOL string is sent at the end of the data. You can change or eliminate the EOL string by using the IOEOL command.
- The EOI line is set active for the last data byte (usually the last EOL character). You can prevent this line from being set by using the IOEOI command to disable EOI.

Example of Formatted Transfers

The following QuickBASIC 4.5 example shows how a program can input and output formatted data:

```
   ⋮
DIM READINGS!(49)
DEVICE& = 723
MAX.ELEM% = 50 : ACT.ELEM% = 0
'Read a maximum of 50 values from device 723
CALL IOENTERA(DEVICE&,SEG READINGS!(0),MAX.ELEM%,ACT.ELEM%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
SUM! = 0
FOR I% = 1 TO ACT.ELEM%
  SUM! = SUM! + READING!(I%)
NEXT I%
DEVICE& = 701
'Send the sum to device 701
CALL IOOUTPUT(DEVICE&,SUM!)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
   ⋮
```

Block-Data Transfers

Some HP-IB devices can send and receive data as a block of bytes—each group of bytes within the block conveys information in an "internal" form that's recognizable to the device and the computer. For example, an integer from −32,768 to 32,767 may be represented by only two bytes, a real number may be represented by four bytes, and a string character may be represented by one byte. The actual number of bytes depends upon the internal designs of the two devices—and it's up to your program to ensure the devices are consistent in interpreting the data.

Block-data transfers are efficient at moving data bytes between a device and the computer's memory because the data isn't interpreted—it's just moved. However, the data is not necessarily recognizable to you by reading the bytes.

Note that for string data, the block transfer simply transfers the characters—it doesn't access or update the string-length parameter or append a terminating character. Note also that you can't transfer data to or from a string using BASIC.

Operation

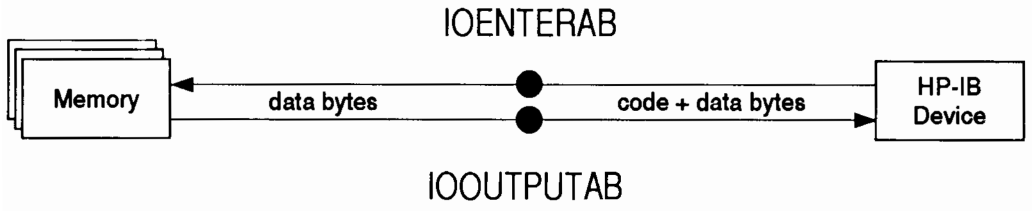
A transfer of block data involves no conversions—data is moved byte-for-byte between the two devices. The Command Library provides transfers for two types of block data:

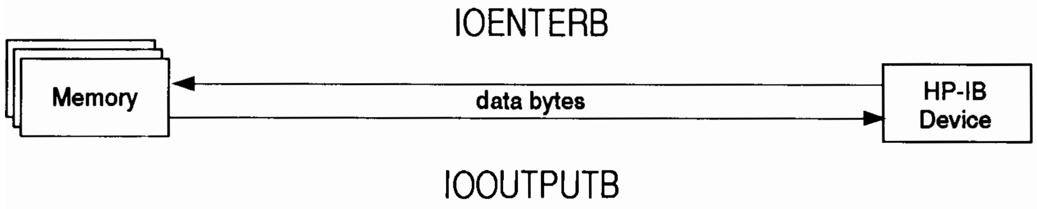
- **Arbitrary-block data.** Block data that has two or more bytes of IEEE-488.2 coding added to define the size of the block. For an output transfer, the size is definite. For an input transfer, the size can be definite, or it can be indefinite (with a termination byte). (See "Arbitrary-Block Data Coding" below for a description of the coding.)
- **Binary-block data.** Block data that has no coding—just data bytes.

If your HP-IB device sends and receives arbitrary-block data (compatible with IEEE-488.2), your program can use the IOENTERAB and IOOUTPUTAB commands to exchange data with the device.

If your HP-IB device sends and receives unformatted, uncoded binary data, your program can use the IOENTERB and IOOUTPUTB commands to exchange data with the device.

The following diagrams represent these transfers.





Optional Conditions

Block-data transfers support the following options:

- Disabling EOI for binary-block input and output. The HP-IB End Or Identify line usually goes active for the last data byte transferred, but you can disable this convention—use the IOEOI command. (Arbitrary-block transfers disregard this option.)
- Changing the EOL string for binary-block output. Data is usually terminated by carriage-return and linefeed characters, but you can change or eliminate these characters—use the IOEOL command. (Arbitrary-block transfers disregard this option.)
- Changing the terminating "match" character for binary-block input. The input of binary data can usually be ended by a linefeed character, but you can change this character or disable this option—use the IOMATCH command. (Arbitrary-block transfers disregard this option.)
- Using byte-swapping for input and output. Block data is usually transferred to and from sequential memory locations, but you can change the order in which memory locations are accessed. (See "Swapping Bytes" below.)
- Enabling DMA for input and output. Data transfers are usually managed by the main processor, but you can specify that the DMA controller route the data directly to and from memory—use the IODMA command. (See "Direct-Memory-Access Transfers" below.)
- Setting the HP-IB transfer timing for output. Data is usually output with standard timing, but you can set high-speed timing—use the IOFASTOUT command.

Ending Arbitrary-Block Input

You can use the IOENTERAB command to enter arbitrary-block data. The input operation ends when one of these conditions occurs:

- The maximum number of bytes specified in the command is received.
- For input of a definite-length block, the number of bytes indicated by the block coding is received.

- For input of an indefinite-length block, a linefeed character is received with the EOI line active. (The linefeed character is not stored in memory.) The EOI option specified by the IOEOI command is disregarded.
- For DMA input in BASIC, a high-priority service request occurs. (See "Service Requests" later in this chapter.)

Ending Arbitrary-Block Output

You can use the IOOUTPUTAB command to send arbitrary-block data with a definite length. At the end of the output operation, no events occur because the coding completely defines the end of the transfer:

- No EOL string is sent. (The EOL string specified by the IOEOL command is disregarded.)
- The EOI line is not set. (The EOI option specified by the IOEOI command is disregarded.)

For DMA output in BASIC, the transfer ends immediately if a high-priority service request occurs. (See "Service Requests" later in this chapter.)

Ending Binary Input

You can use the IOENTERB command to enter binary data. The input operation ends when one of these conditions occurs:

- The maximum number of bytes specified in the command is received.
- A data byte is received with the HP-IB EOI line active. (This byte is stored in memory.) You can prevent the EOI line from ending the transfer by using the IOEOI command to disable EOI.
- The terminating "match" character is received, the EOI option is enabled, and the HP-IB EOI line is active. (The "match" character is not stored in memory.) You can prevent the "match" character from ending the transfer or change the "match" character by using the IOMATCH command—or you can disable the EOI option by using the IOEOI command. You'll normally not use a "match" character because it is likely that within the binary data stream there will be bytes corresponding to the match character.
- For DMA input in BASIC, a high-priority service request occurs. (See "Service Requests" later in this chapter.)

Computer Museum

Ending Binary Output

You can use the IOOUTPUTB command to send binary data. At the end of the output operation these events occur:

- The EOL string is sent at the end of the data. You can change or eliminate the EOL string by using the IOEOL command.
- The EOI line is set active for the last data byte (usually the last EOL character). You can prevent this line from being set by using the IOEOI command to disable EOI.

For DMA output in BASIC, the transfer ends immediately if a high-priority service request occurs. (See "Service Requests" later in this chapter.)

Swapping Bytes

Whenever your program is transferring block data, you must ensure that you set up the transfer command to structure the data properly:

- **Data size.** The sending and receiving devices should treat the data consistently. For example, if the device sends values as two-byte numbers, the computer should use this data as two-byte numbers. For the computer, data size is determined by the variable types used in the program.
- **Data order.** The sending and receiving devices should access data bytes in the same order—or else compensate for differences. For example, most devices send values with the most-significant byte first, and most PC computers interpret data in memory as least-significant byte first. "Byte swapping" enables the computer to swap the order of bytes before they're stored in memory and before they're sent on HP-IB.

The IOENTERAB, IOENTERB, IOOUTPUTAB, and IOOUTPUTB commands provide a flag that specifies byte swapping—the value indicates the size of the data in bytes, from 1 to 8.

The value 1 means one-byte data—no swapping occurs. This is appropriate for string data and for DMA transfers.

The value 4 means four-byte data—groups of four bytes are interchanged before they're stored in memory or sent out. This is appropriate for four-byte data, such as single-precision real numbers.

For arbitrary-block data, only the data bytes are swapped—the IEEE coding isn't involved.

Arbitrary-Block Data Coding

The IOENTERAB and IOOUTPUTAB commands conform to the IEEE-488.2 standard, which defines special coding for arbitrary-block data. This section describes the coding used by these commands.

IOENTERAB accepts input data called "arbitrary-block program data," which can be either definite-length or indefinite-length. The data must be preceded by either (1) three or more characters that define the number of data bytes that follow or (2) two characters that indicate an indefinite number of data bytes (the end is indicated by a linefeed character with the EOI line active).

IOOUTPUTAB sends output data called "definite-length arbitrary-block response data." The data is preceded by three or more characters that define the number of data bytes that follow. (Note that no indefinite-length output is possible.)

The data sequence for *definite-length* data contains four parts (the second character marks the data as definite-length):

1. A # character.
2. One count digit (1 through 9) that indicates how many digits follow to convey the byte count.
3. One to nine digits (as previously specified) that convey the byte count—the number of data bytes that follow.
4. The specified number of data bytes.

The data sequence for *indefinite-length* data contains four parts (the second character marks the data as indefinite-length):

1. A # character.
2. The count digit 0, which indicates an indefinite-length block.
3. Any number of data bytes.
4. A linefeed character with the EOI line active, which indicates the previous data byte was the last data byte.

The following examples show how arbitrary-block data is coded:

`# 1 2` *byte byte*	*One count digit specifying two data bytes.*
`# 3 0 0 2` *byte byte*	*Three count digits specifying two data bytes.*
`# 0` *byte ... byte linefeed/EOI*	*Unspecified number of data bytes.*

Example of Block-Data Transfers

The following QuickBASIC 4.5 example shows how a program can input and output arbitrary-block and binary data with byte swapping:

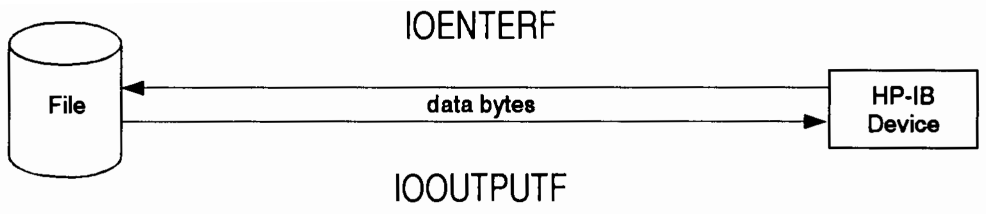
```
:
DIM READINGS#(49)
DEVICE& = 723
SWAP% = 8 : MAX.BYTE% = 50 * SWAP% : ACT.BYTE% = 0
'Read a maximum of 50 binary 8-byte values from device 723
CALL IOENTERB(DEVICE&,SEG READINGS#(0),MAX.BYTE%,ACT.BYTE%,SWAP%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
SUM# = 0
NUM% = ACT.BYTE% / SWAP%
FOR I% = 1 TO NUM%
  SUM# = SUM# + READING#(I%)
NEXT I%
DEVICE& = 701
'Send the sum to device 701 as arbitrary-block data
CALL IOOUTPUTAB(DEVICE&,SEG SUM#,SWAP%,SWAP%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
:
```

File Transfers

HP-IB devices can transfer data directly to and from a file on your computer.

Operation

The IOENTERF and IOOUTPUTF commands transfer data bytes as shown in the following diagram:



Optional Conditions

File transfers support the following options:

- Disabling EOI for input and output. The HP-IB End Or Identify line usually goes active for the last data byte transferred, but you can disable this convention—use the IOEOI command.
- Changing the EOL string for output. Data is usually terminated by carriage-return and linefeed characters, but you can change or eliminate these characters—use the IOEOL command.
- Changing the terminating "match" character for input. The input of file data can usually be ended by a linefeed character, but you can change this character or disable this option—use the IOMATCH command.
- Setting the HP-IB transfer timing for output. Data is usually output with standard timing, but you can set high-speed timing—use the IOFASTOUT command.

Ending File Input

You can use the IOENTERF command to enter file data. The input operation ends when one of these conditions occurs:

- The maximum number of bytes specified in the IOENTERF command is received.
- A data byte is received with the HP-IB EOI line active. You can prevent the EOI line from ending the transfer by using the IOEOI command to disable EOI.
- The terminating "match" character is received. You can prevent the "match" character from ending the transfer or change the "match" character by using the IOMATCH command.

Note If you are transferring binary files, you should turn off character match to make sure the transfer does not end prematurely, since it is likely there will be a byte in the binary data stream that will correspond to the match character.

- A file error occurs, usually meaning the disk is full.

Ending File Output

You can use the IOOUTPUTF command to send the contents of a file to a specified device or interface. After the file is sent, the EOL string is sent and the EOI line is set, if enabled.

Example of File Transfers

The following QuickBASIC 4.5 example shows how a program can transfer file data:

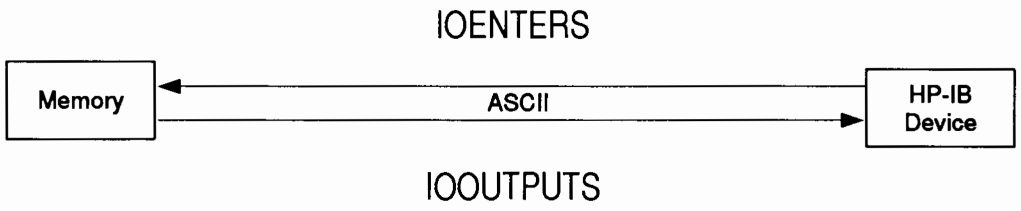
```
:
DEV& = 723
LENGTH& = 10
FILE.NAME$ = "ENTER.DAT"
APPEND% = 0
CALL IOENTERF(DEV&,FILE.NAME$,LENGTH&,APPEND%)
IF PCIB.ERR  NOERR THEN ERROR PCIB.BASERR
:
```

String Transfers

Many HP-IB devices send and receive data as a string of ASCII characters—either digits that represent values, or characters that convey other information. If your program doesn't need this data converted into numeric values, the data can be efficiently transferred as a string of characters.

Operation

The IOENTERS and IOOUTPUTS commands transfer string data and ensure that the computer treats the data as a string variable (a character array). (If you're using Pascal or C, the IOENTERAB, IOENTERB, IOOUTPUTAB, and IOOUTPUTB commands can also transfer string data, but they treat the data as binary data, without accessing or updating the string-length parameter or appending a termination character.) If your HP-IB device sends and receives data as ASCII characters, your program can use the IOENTERS and IOOUTPUTS commands to exchange data with the device. The following diagram represents these transfers.



Optional Conditions

String-data transfers support the following options:

- Disabling EOI for input and output. The HP-IB End Or Identify line usually goes active for the last data byte transferred, but you can disable this convention—use the IOEOI command.
- Changing the EOL string for output. Data is usually terminated by carriage-return and linefeed characters, but you can change or eliminate these characters—use the IOEOL command.
- Changing the terminating "match" character for input. The input of string data can usually be ended by a linefeed character, but you can change this character or disable this option—use the IOMATCH command.
- Enabling DMA for input and output. Data transfers are usually managed by the main processor, but you can specify that the DMA controller route the data directly to and from memory—use the IODMA command. (See "Direct-Memory-Access Transfers" below.)
- Setting the HP-IB transfer timing for output. Data is usually output with standard timing, but you can set high-speed timing—use the IOFASTOUT command.

Ending String Input

You can use the IOENTERS command to enter string data. The input operation ends when one of these conditions occurs:

- A data byte is received with the HP-IB EOI line active. You can prevent the EOI line from ending the transfer by using the IOEOI command to disable EOI.
- The terminating "match" character is received. You can prevent the "match" character from ending the transfer or change the "match" character by using the IOMATCH command.
- The maximum number of characters you specified is received. You specify the maximum number in the IOENTERS command.
- For DMA input in BASIC, a high-priority service request occurs. (See "Service Requests" later in this chapter.)

Ending String Output

You can use the IOOUTPUTS command to send string data. At the end of the output operation these events occur:

- The EOL string is sent at the end of the data. You can change or eliminate the EOL string by using the IOEOL command.
- The EOI line is set active for the last data byte (usually the last EOL character). You can prevent this line from being set by using the IOEOI command to disable EOI.

For DMA output in BASIC, the transfer ends immediately if a high-priority service request occurs. (See "Service Requests" later in this chapter.)

Example of String Transfers

The following QuickBASIC 4.5 example shows how a program can input and output string data:

```
   ⋮
   DEV& = 723 : INFO$ = "1ST1"
   LENGTH% = LEN(INFO$)
   'Send "1ST1" to device 723.
   CALL IOOUTPUTS(DEV&,INFO$,LENGTH%)
   IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
   ⋮
   MAX.LENGTH% = 10 : ACTUAL.LENGTH% = 0
   INFO$ = SPACE$(MAX.LENGTH%)
   'Read a string of 10 characters maximum from device
   CALL IOENTERS(DEV&,INFO$,MAX.LENGTH%,ACTUAL.LENGTH%)
   IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
   INFO$ = LEFT$(INFO$,ACTUAL.LENGTH%)
   ⋮
```

Direct-Memory-Access Transfers

Data transfers are usually managed by the main processor in the computer. But you can specify that the DMA controller route the data directly to and from memory. This is often faster for transferring long strings or large blocks of arbitrary-block data or binary data.

You can use DMA transfers for all types of transfers *except* formatted transfers and file transfers. Therefore you can use DMA for arbitrary-block data, binary data, and string data.

The following table summarizes restrictions for DMA transfers. Any restriction that's violated causes an EUNKNOWN error.

Command	DMA Support	DMA Restrictions
Formatted Transfers:		
IOENTER IOENTERA	Not supported	
IOOUTPUT IOOUTPUTA	Not supported	
Block Transfers:		
IOENTERAB IOENTERB	Supported	Byte-swap flag = 1 (no swapping) Match flag = 0 (disabled)
IOOUTPUTAB IOOUTPUTB	Supported	Byte-swap flag = 1 (no swapping)
String Transfers:		
IOENTERS	Supported	Match flag = 0 (disabled)
IOOUTPUTS	Supported	(None)
File Transfers:		
IOENTERF	Not supported	
IOOUTPUTF	Not supported	

To set up DMA transfers, use the IODMA command. This command requires three parameters:

- **Select code of the interface.**
- **DMA size.** This specifies the minimum number of bytes for which a DMA transfer is used. If the transfer involves at least this number of bytes, a DMA transfer is used. If the transfer involves a smaller number of bytes, a conventional transfer is used. If this value is 0, DMA transfers are disabled.
- **Channel number.** This specifies the DMA channel number to use for DMA transfers. The number must be 2 or 3—channel 3 is recommended because it's less likely to conflict with other usage.

The IODMA command sets up DMA transfers *selectively*. That is, you specify the minimum size of transfers that use DMA. Then "large" transfers use DMA, and "small" transfers don't. A typical value for the DMA size is from 10 to 100 bytes, but the performance of your application may be optimum at a very different DMA size.

Of course you can disable DMA transfers or change the DMA size at any time using the IODMA command again.

The following QuickBASIC 4.5 example shows how a program can set up and use a DMA transfer for binary input:

```
  :
DIM RDGS%(1000)
MAXLENGTH% = 1000 * 2
  :
ISC& = 7 : ADDR& = 5
DEVICE& = 100 * ISC& + ADDR&
DMASIZE% = 10 : CHANNEL% = 3
CALL IODMA(ISC&,DMASIZE%,CHANNEL%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
MATCH% = 0 : EOL$ = "" : SWAP% = 1
CALL IOMATCH(ISC&,EOL$,MATCH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
  :
CALL IOENTERB(DEVICE&,SEG RDGS%(0),MAXLENGTH%,LENGTH%,SWAP%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
  :
```

For DMA transfers in BASIC, the transfer ends immediately if a high-priority service request occurs. The BASIC PEN(1) function returns a value that indicates whether the last transfer was interrupted by a service request: 1 indicates a DMA interrupt, 0 indicates no DMA interrupt. (See "BASIC Service-Request Interrupts" next.)

BASIC Service-Request Interrupts

BASIC languages (except QBasic) support interrupts caused by HP-IB devices—the interrupts are called "service requests" because the device essentially requests the attention of the controller. If the controller enables service-request interrupts, it is automatically notified when such an event occurs, and it can respond as required. (See "Service Requests" in appendix B for more information about the operation of HP-IB service requests.)

If you want your program to know "right away" if a device requests service, you can enable service-request interrupts using the IOPEN Library command. While service-request interrupts are enabled, the computer automatically checks for pending service requests *at the end of each program line*—if a request is pending, the program immediately branches to the routine you specify. Thus, by enabling service-request interrupts, your BASIC program automatically checks for requests without actually branching to a status-checking routine—it branches only after a request occurs.

You should be aware that the Command Library implements service-request interrupts using the BASIC functionality of the PEN commands, which were designed for interaction with a light pen. If your program requires the use of a light pen, the program shouldn't enable service-request interrupts—instead, it can use the IOSTATUS Library command at certain times to find out whether a request is waiting.

If you want your BASIC program to check HP-IB devices only at a certain part of the program, you don't need to use service-request interrupts—you can use the IOSTATUS Library command to find out whether a device has requested service since you last checked.

Enabling and Disabling Service-Request Interrupts

To enable service-request interrupts, your program must do the following:

1. **Define the interrupt branching.** Use the ON PEN command in BASIC to define the branching to perform when an event occurs.
2. **Enable the event to be logged.** Use the PEN ON command in BASIC to allow the program to recognize and respond to service-request interrupts.

3. **Set up the interrupt event.** Use the IOPEN Library command to define the select code of the HP-IB interface and the priority of the interrupt.
4. **Include a service routine.** Include commands that perform the desired action whenever a service-request interrupt occurs. (See "Servicing a Request" below.)

The following QuickBASIC 4.5 example shows how a program can enable service-request interrupts and branch to a routine when one occurs:

```
:
ISC& = 7 : PRIORITY% = 0
ON PEN GOSUB SRQ                 'define interrupt branching
PEN ON                           'enable SRQ event to be logged
CALL IOPEN(ISC&,PRIORITY%)       'set up interrupt event
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
:
'main part of program here
:
SRQ:                             'service routine
:
'service-request routine here
:
RETURN
```

To disable or suspend service-request interrupts, your program can do either of the following:

- **Disable requests.** Use the PEN OFF command in BASIC to prevent the program from logging and responding to service requests. Requests that occur are discarded.
- **Suspend requests.** Use the PEN STOP command in BASIC to temporarily prevent the program from responding to service requests. However, requests are still logged, and they're processed when PEN ON is next executed.

Although your program may disable or suspend service-request interrupts, the IOPEN command has set up the interface to look for service requests—and this continues to happen, even after PEN OFF or PEN STOP. For this reason, if the IOPEN command sets service-request interrupts to interrupt DMA transfers, they'll be interrupted even though service-request interrupts are disabled or suspended. (See "Interacting with DMA Transfers" below.)

The setup provided by IOPEN remains in effect until one of the following commands is executed. If you want service-request interrupts to be enabled after one of these commands, execute IOPEN again.

- Another IOPEN command, which can redefine the priority (the select code shouldn't change).
- An IORESET command, which resets the HP-IB interface.
- A SHELL command, which temporarily reverts to MS-DOS, then clears the interrupt setup.

Servicing a Request

Your service-request routine should take action according to the type of situation that occurred. (In addition, it should perform some standard service-request functions described below.)

For example, if your system has only one device and it will request service only when it has data available, then your service-request routine simply needs to clear the service request and enter data from the device.

However, if your system has several devices that can request service, or if service can be requested for several reasons, then your service-request routine should determine the device and reason, then take the required action.

Your service-request routine should normally include commands to do the following:

1. **Suspend requests.** BASIC automatically suspends service-request interrupts when the interrupt branch occurs—this prevents repeated branching from a service request. The program automatically performs a PEN STOP. PEN STOP lets the program remember requests that occur while the service-request routine is executing. You can include PEN OFF to stop and not remember incoming requests.
2. **Poll devices.** Use IOSPOLL to read the status from each device that could be requesting service. This lets you find the device that's requesting service. (Your routine could use IOPPOLL to help find the device faster.) IOSPOLL also clears the request—the same request won't be serviced again.
3. **Enable requests.** If you haven't stopped service-request interrupts with PEN OFF, BASIC automatically reenables interrupts when the routine

returns—this lets the program resume servicing requests. To do this, the program automatically performs a PEN ON.

The following QuickBASIC 4.5 example shows how a program can process a service request (assuming only one device can request service):

```
:
ISC& = 7
ADDR& = 4
DEVICE& = 100 * ISC& + ADDR&
:
SRQ: 'service request routine--auto PEN STOP suspends interrupts
CALL IOSPOLL(DEVICE&,STATUS%) 'clears request
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
PRINT "Status:",STATUS%
:
RETURN                            'auto PEN ON enables interrupts
```

Interacting with DMA Transfers

Your program can set up service-request interrupts with one of two priority levels:

- High priority—they *can* interrupt DMA data transfers.
- Low priority—they *can't* interrupt DMA data transfers.

A program can enable service requests to stop DMA data transfers—for example, another critical device can receive immediate service even if a DMA transfer is in progress.

However, if your program wants to interrupt DMA transfers in one section but not in another section, it must execute IOPEN to change the priority. It can't use PEN OFF and PEN STOP to disable service-request interrupts—they don't stop the interface from looking for service requests. If the IOPEN priority is high, a service-request interrupt will interrupt a DMA transfer regardless of whether requests are processed.

The following QuickBASIC 4.5 example shows how a program can set service-request interrupts active for the first DMA transfer and inactive for the second transfer:

Computer Museum

```
⋮
DIM RDGS&(10000)
ISC& = 7 : ADDR& = 100 * ISC& + 4
⋮
SIZE% = 50 : CHAN% = 3 : EN% = 1
CALL IODMA(ISC&,SIZE%,CHAN%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
ON PEN GOSUB SRQ
PEN ON                                'enable SRQ processing
CALL IOPEN(ISC&,EN%)                  'allow DMA interrupts
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
MAX% = 10000 : LENGTH% = 0 : SWAP% = 1
CALL IOENTERB(ADDR&,SEG RDGS&(0),MAX%,LENGTH%,SWAP%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
⋮
PEN OFF                               'disable SRQ processing
EN% = 0
CALL IOPEN(ISC&,EN%)                  'no DMA interrupts!
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
MAX% = 1024 : LENGTH% = 0
CALL IOENTERB(ADDR&,SEG RDGS&(0),MAX%,LENGTH%,SWAP%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
⋮
SRQ:                                  'service request routine
CALL IOSPOLL(ADDR&,STATUS%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
IF PEN(1) = 1 THEN PRINT "DMA interrupted."
RETURN
```

The second IOPEN command sets the low interrupt priority. If this command were omitted, a service request occurring during the second DMA transfer would interrupt the transfer (although it wouldn't be processed by the service-request routine).

Your service-request routine can use the PEN(1) function to determine whether a DMA transfer was ended by the interrupt. If a DMA transfer was ended, PEN(1) returns the value 1; otherwise, it returns the value 0. (Normally, the PEN(1) function returns the x coordinate of the light pen.)

Pascal and C Service-Request Interrupts

HP-IB protocol provides that HP-IB devices may request attention from the controller by issuing a "service request"—a request to interrupt the normal program flow. Your Pascal or C program can check for service requests at any time by using the IOSTATUS Library command.

The program should check the interface status as often as necessary to provide adequate response time. If you need immediate response throughout your program, you may want to include a branch to a status-check routine between most program lines.

The status routine should execute IOSTATUS (condition 1). The returned status value indicates whether service has been requested or not. If service *has* been requested, the program can branch to a service-request routine. The service-request routine can use IOSPOLL to read the status from each device that could be requesting service. This lets you find the device that's requesting service. (Your routine could use IOPPOLL to help find the device faster.) IOSPOLL also clears the request—the same request won't be detected again.

The following example shows how a Turbo Pascal program can check for a service request (assuming only one device can request service):

```
:
PROCEDURE check_srq;
CONST
  isc = 7;
  addr = 4;
  condition = 1;
VAR
  device : LONGINT;
  error : INTEGER;
  status : INTEGER;
  err : INTEGER;
BEGIN
  device := isc * 100 + addr
  err := IOSTATUS(isc,condition,status);
  IF err <> NOERROR THEN errorcheck(err);
  IF status = 1 THEN BEGIN
    err := IOSPOLL(device,status);
    IF err <> NOERROR THEN errorcheck(err);
    :
  END;
END;
:
```

Processing I/O Errors

During program execution, errors may occur as a result of I/O operations. For example, a device may not respond, a timeout may occur, or you might specify an incorrect select code or some invalid parameter. Because there is no built-in error reporting procedure for Library commands, you should include error handling code in your program. Otherwise, errors go untreated and may cause unpredictable results. To help you provide the necessary code, this section explains general error handling. All Command Library errors are described in appendix A, "Error Descriptions."

Error Message Mnemonics

As an aid to error testing, each error has an associated mnemonic variable. The mnemonics are declared in the Command Library file appropriate for your programming language. You can use the mnemonic variables to test for the occurrence of errors. For example, error 0—no error occurred—has the mnemonic NOERR.

In BASIC, the following statement would cause program execution to branch to subroutine ERRCHECK if an error occurred.

```
IF PCIB.ERR <> NOERR THEN GOSUB ERRCHECK
```

In Pascal, the following statements would perform the Errcheck routine, which can compare the error status indicated by `err` and NOERR, then take action if an error occurred.

```
err := IOENTER (709,reading);
Errcheck (err);
```

All Command Library errors and their mnemonics are listed in appendix A, "Error Descriptions."

BASIC Error Variables

To assist you in detecting and handling errors in BASIC programs, the Command Library provides three error status variables. (These variables are not available for Pascal or C.)

PCIB.ERR	This is a return status variable indicating whether an error was detected in the last Library CALL statement. It is good practice to check the value of PCIB.ERR after every CALL statement. If PCIB.ERR = 0, the command terminated without error.
PCIB.ERR$	When PCIB.ERR is nonzero, this string variable contains an error message corresponding to the value of PCIB.ERR.
PCIB.GLBERR	This variable indicates if an error occurred anywhere in the program. If PCIB.GLBERR = 0, all preceding

Library calls completed without error. Otherwise, PCIB.GLBERR contains the most recent error value. To locate the exact statement that caused the error, you must perform more detailed error checking.

These three status variables reflect some HP-IB—specific errors (100000 to 100009). They do not, however, reflect spelling errors or parameter passing-errors such as passing the wrong number or type of parameters in the CALL statement.

Caution

Spelling errors and parameter-passing errors are not reflected by the error status variables. If the wrong number or type of parameters are passed in a CALL statement, the resulting error may require the system to be restarted. Since your program may contain such errors, it is a good idea to always save your program before you run it.

Error Reporting

Your program should check for errors after each Library command. Since a program normally has more than one CALL statement, you can write a single error handling routine to use after all CALLs. Examples are included for each type of programming language in chapters 2 through 6.

For example, you may want to create an error handling routine that treats various errors (for example, timeout or invalid select code) differently, or that resumes program execution after an error is logged or acknowledged. Whether you use the error handling routine provided or write your own, you will save time during program testing if you check for errors after each Command Library command.

Addressing Library Commands

The first parameter of each Library command specifies an interface select code or a device bus address. Some commands allow only a select code (for example, IOABORT). Some permit only a device address (for example, IOSPOLL). And some permit either (for example, IOCLEAR). If you specify a select code, the command is directed to the interface, and no bus addressing is performed prior to the transfer of data or commands. If you specify a device address, devices on the bus are first addressed using the following sequences.

For entering data:

1. Unlisten.
2. My Listen Address.
3. Talk Address of the target device.

For outputting data:

1. My Talk Address.
2. Unlisten.
3. Listen Address of the target device.

These sequences conform to the IEEE 488.2 specifications. The addressing is followed by a data transfer or a bus command, depending on the command.

A select code is a value in the range 0 to 99—only 1 through 16 are valid select codes, however, and there is only one valid select code for each HP-IB interface in your computer. If you specify a nonexistent select code, an error results.

Note that the *select_code* and *device_address* parameters must be the appropriate type for your programming language. If an integer variable is required, it is a "long" or "four-byte" integer—this provides the range required for extended addressing, described below.

Basic Addressing

A basic device address is composed of the select code and the primary bus address of a device. It is calculated as

(*select code* × 100) + *primary bus address*

A valid primary bus address is in the range 0 to 30. Address 30 is reserved as the address of the controller.

Extended Addressing

You can use extended talker and listener functions of the Library by using extended addressing. To specify an extended address, calculate the device address as

(*select code* × 10000) + (*primary bus address* × 100) + *secondary address*

A secondary address may be in the range 0 to 31. You can use extended addressing with any command in which the first parameter may be a device address.

As an example, consider a system with select code 7 that has a device with primary address 9 and secondary address 15. Each of these QuickBASIC 4.5 IOCLEAR commands is valid.

```
ISC& = 7
CALL IOCLEAR (ISC&) 'Select Code Only
DEVICE& = 709
CALL IOCLEAR (DEVICE&) 'Primary Address
DEVICE& = 70915
CALL IOCLEAR (DEVICE&) 'Secondary Address
```

System and Active Control

If you have an HP 82335 HP-IB interface, you can use the pass control capabilities of the Command Library. This section describes system and active control, and how to transfer control. It also describes a DOS program that controls the system control status, and the capabilities of the Command Library while it is not a controller. If your HP-IB interface is the older HP 82990 or HP 27209 board, non-controller capabilities will not function properly.

Description of System and Active Control

The *system controller* is the primary controller of an HP-IB system. There can be only one system controller in a system. The device that is system controller will always be system controller.

The *active controller* is the device in the system that currently controls the ATN line and sends bus commands to all other devices. It is the device currently in charge of the interface and it controls all bus traffic. There can be only one active controller at a time in a system, but active control may be passed from one device to another.

When a system is first started, the device that is system controller is also the active controller. The active controller can, however, pass on active controller responsibilities to any other device on the bus capable of being an active controller.

System Controller

The system controller has exclusive use of the *Interface Clear* (IFC) line and the *Remote Enable* (REN) line of the HP-IB bus.

The IFC line is used to instantly regain active control if it has been passed to another device, abort all bus activity, unaddress all other devices, and disable serial poll. It is like a master reset line. The Command Library can assert IFC using either IORESET, IOABORT, or IOTAKECTL with priority 3. IORESET has other side effects, such as restoring EOI, EOL, timeout, and others to their power on default values.

The REN line is used to place devices in remote programming mode. It can be set using the Library Command IOREMOTE and cleared using IOLOCAL.

Active Controller

The active controller is in charge of controlling all bus traffic and sending all bus commands to devices. The active controller determines which device will talk and which device(s) will listen.

Using the Command Library, active control can be passed to another device using the IOPASSCTL command. When you want the HP 82335 interface to become the active controller again, use the IOTAKECTL command. If you want to request service from the currently active controller, use the IOREQUEST command to assert SRQ and set up a response byte.

Each of these commands is described in the appropriate reference chapter of this manual.

For your reference, several sample programs are located in an archive file named PASSEXMP.EXE on the source disks. To unarchive these programs, use the DOS cd command to change your current directory to where you want the sample programs copied. Then, if the source disk is in drive A:, type `A:PASSEXMP` at the DOS prompt to copy the sample programs to your current directory.

SYSCTL.EXE Program

This is a DOS program that controls the system controller status of the HP 82335 interface.

Syntax

`C:\ > SYSCTL` *select_code status*

select_code	specifies the interface select code.
status	specifies if system control should be turned off or on. If this parameter is a zero, the interface will be made the non-system controller. If it is non-zero, the interface will be the system controller.

Examples

```
C:\  > SYSCTL 7 0
```

This will make the interface at select code 7 a non-system controller.

```
C:\  > SYSCTL 5 1
```

This will make the interface at select code 5 a system controller.

Comments When you first turn on your computer, the HP 82335 interface

will be set to system controller.

This utility is located on both the 3.5-inch Disk 1—Install and 5.25-inch Disk 1—Install disks.

See the comment about Ctrl-C and Ctrl-Break in the IOPASSCTL section for important information.

Do *not* use the printer/plotter driver (HPIB.SYS) when the HP-IB board is non-system controller.

When the Command Library is Not Controller

The HP 82335 Command Library and interface can function as system or non-system controller as well as active or non-active controller.

The HP 82335 Command Library and interface can read and write data using any of the enter or output statements while non-active controller. They cannot, however, specify a device address, as this is the responsibility of the active controller.

The HP 82335 Command Library can respond to a serial poll, but cannot respond to a parallel poll.

Detecting HP-IB Cards Programmatically

The easiest way for your programs to automatically find HP-IB cards is to call `IOSTATUS(X,8,status)` in a loop. The variable 'x' should start at 1 and end at 16. If the return value is NOERR, there is a card at location x. If the return value is ESEL, there is no card at location x.

Where to Go Next

The general information in this chapter supplements the detailed information in the following chapters:

- If you're programming with GW-BASIC or a similar language, go to chapter 2, "GW-BASIC Programming."
- If you're programming with QuickBASIC, QBasic, Compiled BASIC, BASIC PDS, or a similar language, go to chapter 3, "QuickBASIC Programming."
- If you're programming with Pascal, go to chapter 5, "Pascal Programming."
- If you're programming with the C language, go to chapter 6, "C Programming."
- If you're programming with FORTRAN, go to the section named "Using Microsoft FORTRAN With C" in chapter 6.

Additional information is available in the appendixes:

- A description of Command Library errors is included in appendix A, "Error Descriptions."
- A brief description of HP-IB is included in appendix B, "Summary of HP-IB."
- A list of reserved names is included in appendix C, "Reserved Names."

2

GW-BASIC Programming

Introduction

This chapter explains how to use the HP-IB Command Library for GW-BASIC programming.

Supported versions of GW-BASIC and similar languages are listed on the *Supported Languages* sheet included with the Command Library. For example, you can use certain versions of Vectra BASIC and GW-BASIC on an HP Vectra computer. For an IBM or compatible PC, you can use certain versions of BASICA.

Note

If you are having difficulties running GW-BASIC on a Vectra 486 (even without loading the HP-IB Command Library), try turning cache off by running the *Easy Config* configuration utility.

This chapter contains several sections describing how you can use the Command Library with GW-BASIC:

- Copying the necessary Library files to a work disk.
- Writing a sample program using several Library commands. Your HP-IB Command Library disk contains a similar version of this program. You can run it on your system to observe the results.
- Processing errors.
- Learning about parameters for Library commands.
- Checking example programs. Two listings at the end of this chapter show how you can use Library commands in GW-BASIC programs.

Detailed syntax information for the commands as they're used with GW-BASIC is included in chapter 4, "BASIC Reference."

Copying Files

To begin programming in BASIC, you must copy the BASIC Library files to your work disk.

The HP-IB Command Library disk contains an INSTALL program that copies the GW-BASIC Library files to your system for you.

To use INSTALL:

1. Insert the Library disk into your flexible disk drive—if you're using 5.25-inch disks, use the disk labeled "Disk 1—Install."
2. Run INSTALL by typing

   ```
   a:install Enter
   ```

3. Follow the instructions displayed on your screen. When you have successfully completed the instructions, the following files are copied:

   IBHPIB.LIB
   SETUP.BAS
   NODOC.BAS
   EXAMPLE.BAS
   IBAS.BAT
   VIBAS.BAT
   GWMETER.BAS
   GWSCOPE.BAS

4. When asked, allow the program to change your AUTOEXEC.BAT file—it adds a line to define the PCIB environment variable. Press Y. (If you press N, you can edit the file yourself later.)
5. Remove the Command Library disk.
6. If you allowed INSTALL to change your AUTOEXEC.BAT file, restart your system by pressing Ctrl-Alt-Del (hold Ctrl and Alt, then press Del).

Also make sure you copy all the necessary files from your BASIC interpreter to your system. Refer to your BASIC manual for details.

Setting the Environment

In preparation for BASIC programming, you must set some environment characteristics. Follow these steps to assign the PCIB environment variable, set the search path, and select BASIC:

Note

For convenience you can set up your system to automatically perform the next two steps. Simply edit the AUTOEXEC.BAT file and add the next two commands to the file. They'll be executed the next time you start your system.

1. Assign the PCIB environment variable. It should correspond to the destination where you copied the Library files earlier. (If you allowed INSTALL to change the AUTOEXEC.BAT file above, the PCIB variable is automatically set to the proper value.) For example, type

   ```
   set pcib=c:\hpib (Enter)
   ```

   This variable identifies the location of the Library. If this variable is not set correctly, you will get a "file not found" error when you run a BASIC program.

   If you get an "out of environment space" message, type `set` to see your current environment variables. To create more environment space, reduce the number of variables and reboot, or use COMMAND.COM with the /E: option to increase the environment space.

2. In preparation for loading BASIC, set PATH to reflect the location of the Command Library files and the BASIC files. To find out the current PATH variable, you can type

   ```
   path (Enter)
   ```

   If the locations of the Library and BASIC are not in the current PATH, retype the current path and add the Library and BASIC directories. For example, if BASIC is in the BAS directory on the hard disk and the Command Library is in the HPIB directory, you can add them to the current PATH by typing

   ```
   path=c:\;c:\dos;c:\bas;c:\hpib (Enter)
   ```

3. Load BASIC into memory:
   - **For GW-BASIC and Vectra BASIC** type

     `vibas` (Enter) (for GW-BASIC or Vectra BASIC on HP Vectra)
   - **For BASICA** type

     `ibas` (Enter) (for BASICA on IBM or compatible)

   If you want to load and run an existing program when you load BASIC into memory, type the file name after VIBAS or IBAS. For example,

   ```
   vibas myprog (Enter)
   ```

   The language you choose depends on your computer. Refer to your computer manual for additional language installation instructions.

Note

If you try to execute GWBASIC or BASICA to load BASIC (instead of executing VIBAS or IBAS), you may encounter problems using the BASICA SHELL command or accessing all required memory.

Programming in GW-BASIC

For GW-BASIC programming, the Library is implemented as a series of assembly language subroutine calls. To access the subroutines, your application program must include the information from the SETUP.BAS Library file. This file acts as a header for your application program to provide entry points into the subroutine calls. To save memory, you can use the uncommented version of SETUP.BAS called NODOC.BAS supplied on the Library disk.

The following diagram shows how your application program merges with the SETUP.BAS file.

```
1
2     These lines contain the
3     SETUP.BAS file information
4     that calls the necessary
5     command subroutines  ...
:
999

1000
1001  These lines contain
1002  your application
1003  program  ...
:
9999
```

There are several ways to combine your application program with the SETUP.BAS information:

- Write your program, then merge SETUP.BAS into it.
- Start with SETUP.BAS, then add your program to it.

You can write your program in a separate file, then merge SETUP.BAS into it. With this method, your program should begin at line 1000. When you are ready to merge, load your program and type

```
merge "setup" Enter
```

Since SETUP.BAS starts at line 5 and your program starts at line 1000, this merges SETUP.BAS into the beginning of your application program. You can save the result under your application program name. For example

```
save "program" Enter
```

Or you can load SETUP.BAS and write your application program within it. Again, start line numbering at 1000, after the SETUP.BAS program lines. In this case, you do not have to merge anything, but you will want to save the result under a new filename so you don't overwrite SETUP.BAS. For example

```
load "setup" Enter
```

Start your application at line 1000. When you're finished, save the result

```
save "program" Enter
```

In the following example, the BASIC program is written within the SETUP.BAS file.

Writing a BASIC Program

In an application program, you typically use the Library commands in the following manner to execute an operation:

1. Set up the required variables.
2. Perform the operation.
3. Test to see if the operation completed successfully.

In this example, you follow these steps to program two instruments—an HP3325A Synthesizer/Function Generator and an HP3456A Digital Voltmeter. You program the source to output a 2-V rms signal, swept from 1 kHz to 10 kHz. You program the DVM to take 20 readings from the signal and output them to an array. Finally, you display the readings on the screen.

From BASIC, begin by loading SETUP.BAS.

```
load "setup" Enter
```

Generally, you start line numbering at 1000, after the subroutine calling information.

```
auto 1000 Enter
```

Here, the line numbers have been set to coincide with those of the example file EXAMPLE.BAS on the Library disk.

1. Define some working variables.

```
1070 OPTION BASE 1
1080 MAX.ELEMENTS = 20
1090 DIM READINGS (MAX.ELEMENTS)
1100 ACT.ELEMENTS = 0
1110 CODES$ = SPACE$(50)
```

1070 Set the array base to 1 so array element numbering begins with 1 instead of 0.

1080 Define a maximum-readings variable (MAX.ELEMENTS) so you can easily change this parameter when desired.

1090 Dimension an array (READINGS) to hold the readings taken by the voltmeter.

1100 Set ACT.ELEMENTS, the actual number of elements read by IOENTERA, to 0. ACT.ELEMENTS is an array parameter that must be dimensioned or initialized prior to its use in IOENTERA.

1110 Initialize a string (CODES$) to hold a sufficient number of instrument programming codes.

Note

If your program chains to other programs, you will need COMMON declarations to pass parameters to those programs. Also, you must call DEF.ERR upon entering a chained program to set up pointers to the Library error variables. The SETUP.BAS file in appendix B contains information on these topics. For more information on chaining, see your BASIC manual.

---

2. Initialize the bus and instruments.

```
1150 ISC = 7
1170 DVM = 722
1200 SOURCE = 717
1230 CALL IORESET (ISC)
1240 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1250 TIMEOUT = 5
1260 CALL IOTIMEOUT (ISC, TIMEOUT)
1270 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1280 CALL IOCLEAR (ISC)
1290 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

1150 Set the interface select code variable ISC to 7.

1170 Set the voltmeter address variable DVM to 722.

1200 Set the signal source address variable SOURCE to 717. (This example assumes select code 7, voltmeter address 22, and source address 17.)

1230 Set the interface to its default configuration.

1260 Define a system timeout of 5 seconds.

1280 Perform IOCLEAR to put all instruments into a known, device-dependent state.

Note

This program includes an error checking line after each Library command:

```
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

If there is an error (that is, the error variable PCIB.ERR does not equal NOERR), this line calls the SETUP.BAS error handling routine. This routine prints error information and stops the program. See "Processing I/O Errors" in chapter 1 and "BASIC Error Handling" later in this chapter for more information.

---

3. Program the instruments.

```
1350 CODES$ = "RF2 FU1 ST1KH SP10KH MF1KH AM2VR TI5SE"
1360 LENGTH = LEN (CODES$)
1370 CALL IOOUTPUTS (SOURCE,CODES$,LENGTH)
1380 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1410 '
1420 CODES$ = "H SM004 F2 R4 FL0 Z0 4STG 20STN RS1 T4"
1430 LENGTH = LEN(CODES$)
1440 CALL IOOUTPUTS (DVM,CODES$,LENGTH)
1450 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

1350 Define the programming codes string CODES$ to hold:

RF2—select the rear panel signal output.
FU1—select the sine wave function.
ST1KH—select a starting frequency of 1 kHz.
SP10KH—select a stopping frequency of 10 kHz.
MF1KH—select a marker frequency of 1 kHz.
AM2VR—select an amplitude of 2 V rms.
TI5SE—select a sweep time of 5 seconds.

Computer Museum

1370 Use the IOOUTPUTS command to send the programming codes to the source with the proper length parameter.

1420 Define the programming codes string CODES$ to hold:

H—software-reset the voltmeter.

SM004—set the service request mask to enable the voltmeter to set the interface SRQ line when it finishes taking readings (when the Data Ready bit of the serial poll response byte is set).

F2—select the AC volts function.
R4—select the 10 volt range.
FL0—turn off filtering.
Z0—turn off auto zero.
4STG—select the four-digit display.
20STN—take 20 readings.

RS1—turn on reading storage.
T4—select trigger hold.

1440 Send the programming codes to the voltmeter with the proper length parameter.

4. Trigger the instruments.

```
1490 CALL IOTRIGGER(DVM)
1500 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1510 CODES$ = "SS"
1520 LENGTH = LEN(CODES$)
1530 CALL IOOUTPUTS (SOURCE,CODES$,LENGTH)
1540 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

1490 Execute the IOTRIGGER command for the voltmeter.

1510 Define the programming codes string to hold the source's trigger code.

1530 Send the programming codes to the source with the proper length parameter.

These lines demonstrate that some instruments respond to an HP-IB trigger command, while others must be triggered with instrument-specific programming codes.

5. Wait for the voltmeter to finish reading.

```
1580 SRQ = 1
1590 CALL IOSTATUS (ISC, SRQ, STATUS)
1595 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1600 IF STATUS = 0 THEN GOTO 1590
1610 CALL IOSPOLL (DVM, RESPONSE)
1615 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1620 IF (RESPONSE AND 68) <> 68 THEN GOTO 1590
```

1580 Define the interface condition whose status is being checked. In this case, check for condition 1—is the SRQ line set?

1590 Execute the status command and return the result in STATUS.

1600 As long as STATUS is 0, the SRQ line is not set, indicating that the voltmeter is not finished taking readings.

1610 As soon as STATUS changes to 1, perform a serial poll on the voltmeter to learn which of its conditions, if any, set the SRQ line. The serial poll also clears the SRQ.

1620 The result of the serial poll is the status byte of the voltmeter, returned in RESPONSE. Compare RESPONSE and the value 68—the sum of the Request Service bit (64) and the Data Ready bit (4). If these bits are set, continue because the voltmeter is finished. If they are not set, perform the status check again.

6. Enter the readings into an array and print them.

```
1710 CODES$ = "S01 -20STR RER"
1720 LENGTH = LEN(CODES$)
1730 CALL IOOUTPUTS (DVM,CODES$,LENGTH)
1740 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1741 '
1745 STATE = 0
1750 CALL IOEOI (ISC,STATE)
1760 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1770 CALL IOENTERA (DVM,READINGS(1),MAX.ELEMENTS,ACT.ELEMENTS)
1780 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1810 '
1820 PRINT "THE READINGS ARE: "
1830 FOR I = 1 TO ACT.ELEMENTS
1840 PRINT I, READINGS(I)
1850 NEXT I
1860 '
1870 END
```

1710 Define the programming codes string to direct the voltmeter to output its stored readings:

SO1—turn on system output mode.
−20STR—unstore the 20 readings from register R.
RER—recall (readout) the 20 readings.

1730 Output the programming codes to the voltmeter with the proper length parameter.

1750 Disable the EOI mode so reading won't terminate after entering only one value.

1770 Enter the voltmeter readings into the READINGS array starting at the first element. Include the maximum and actual length parameters.

1840 Print the readings.

Saving the BASIC Program

When you have finished writing the program, save it:

1. Press Ctrl-C to end auto line numbering.
2. Save the program as an executable BASIC file in ASCII. For example, type

```
save "program",a Enter
```

Your program, with the SETUP.BAS material, now resides as an ASCII file in the Library directory as PROGRAM.BAS. Saving the file in ASCII format (the "A" parameter) allows you to perform a subsequent MERGE.

Running the BASIC Program

When you're ready to run the program, connect the SIGNAL output of the source to the VOLTS input of the DVM. (Include a 50-ohm load in this line to ensure proper readings.) Use HP-IB cables to connect the instruments to your computer.

To execute your program from BASIC, load it and type RUN. Watch the display on the function generator. You will see the various functions (sine wave, AC volts, sweep time) displayed as they are programmed. The voltmeter displays its operation as well—you can watch it take readings, store them, and output them to the READINGS array. As the program ends, it displays the readings on your screen.

Note If the name of a Library command is misspelled, or if the number or types of parameters is wrong, system restart may be required to recover.

BASIC Error Handling

General information about Command Library errors, BASIC error variables, and how to process errors are contained in "Processing I/O Errors" in chapter 1 and in appendix A, "Error Descriptions."

If you don't need to write a special error-poocessing routine for your program, you can use the one provided in SETUP.BAS.

```
385  '  Error handling routine
390  '
395  IF ERR=PCIB.BASERR THEN GOTO 410
400  PRINT "BASIC error #";ERR;" occurred in line ";ERL
405  STOP
410  TMPERR = PCIB.ERR
415  IF TMPERR = 0 THEN TMPERR = PCIB.GLBERR
420  PRINT "HPIB error #";TMPERR;" detected at line ";ERL
425  PRINT "Error: ";PCIB.ERR$
430  STOP
```

This error handling routine is based on the BASIC function ERROR and its associated variables ERR and ERL. (If you need more details about ERROR, ERR and ERL, refer to your BASIC manual.) The variable PCIB.BASERR (value 255) provides a mechanism for differentiating between a BASIC error and an HP-IB Command Library error. If ERR = PCIB.BASERR, the error is generated by the Command Library—otherwise, it's a BASIC system error.

The above routine prints the type of error (BASIC or HP-IB), the error number, the error message, and the line on which the error was detected. Then, it stops the program. An ON ERROR statement (see

SETUP.BAS) defines a branch to the error handling routine for any BASIC or program-initiated call to the ERROR function.

Command Library Parameters

This section presents information about Command Library parameters as they are used with GW-BASIC.

Passing Parameters

In BASIC, all parameters used in CALL statements must be passed by reference. That is, you must use variable names as parameters—literals or expressions are not permitted. For example, this statement is valid

```
1050 ISC = 7
1060 CALL IOCLEAR (ISC)
```

but this one is invalid

```
1050 CALL IOCLEAR (7)
```

Parameter Types

Several types of variables are used to describe parameters to Library command calls in chapter 4, "BASIC Reference."

Numeric Variable

A numeric variable is a single-precision real number variable. It is distinguished from other identifiers either by "!" appended to the variable name, or by no suffix appended at all. The valid range for numeric variables is approximately 2×10^{-39} to $2 \times 10^{+38}$ (negative or positive). Single-precision real numbers have approximately seven digits of accuracy. Note that integers and double-precision real numbers may not be used as parameters—except as allowed for data with IOENTERAB, IOENTERB, IOOUTPUTAB, and IOOUTPUTB (see "Any Type of Array" below).

Valid Numeric Variables:	ISC DEVICE.ADDRESS READING!
Invalid Numeric Variables:	REASON% A$ AREA#

String Variable

A string variable is a variable that can contain any valid sequence of 0 or more ASCII characters. It is identified by a "$" appended to its identifier. The length of a string variable is not fixed, but may be anywhere from 0 (the null string) to 255. String variables should be initialized with SPACE$ before data is entered with an IOENTERS command. Otherwise, if the current length is zero, no data will be transferred.

Numeric Array

A numeric array is an array of single-precision real numbers. Arrays are declared using the DIM statement. Although the theoretical maximum size for a BASIC array is 32,767, the actual limit is approximately 14,000 elements depending on the length of your program. GW-BASIC does not permit the use of unsubscripted array names as parameters to CALL statements. Therefore, when you use arrays in Library function calls, use the first element of the array to be accessed as the parameter. For example

`VALUES(0)`	*Starts at element 0.*
`VALUES(5)`	*Starts at element 5.*

Be sure the number-of-elements parameter value does not exceed the number of elements available in the array. For data output, the Library continues through memory sending the contents of the individual cells. For data input, the Library continues through memory, writing over existing data or programs.

Any Type of Array

Any type of array can be a numeric array (single-precision real array), an integer array, a double-precision real array, or other type of numeric array.

(The array can't be a string.) It indicates the place to start reading or storing data.

BASIC does not permit the use of unsubscripted array names as parameters to CALL statements. Therefore, when you use arrays in Library function calls, use the first element of the array to be accessed as the parameter. For example

`VALUES(0)`	*Single-precision real—starts at first element.*
`DATA#(5)`	*Double-precision real—starts at element 5.*
`FLAGS%(0)`	*Integer—starts at first element.*

Be sure the number-of-bytes parameter value does not exceed the number of bytes available in the array—the number of elements times the number of bytes per element. For data output, the Library continues through memory sending the contents of the individual cells. For data input, the Library continues through memory, writing over existing data or programs.

Example Programs

Oscilloscope Example

The following program is written in GW-BASIC. The program uses two devices: HP 54601A digitizing oscilloscope (or compatible scope) and a printer capable of printing HP Raster Graphics Standard, such as a ThinkJet printer.

The program tells the scope to take a reading on channel 1 and send the data back to this program. Then it prints some simple statistics about the data. 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 HP-IB transaction.

Things to note about this program:

- Note the use of the IOENTERAB command. This command will read an arbitrary block of data as defined in IEEE-488.2. IOENTERAB can read either definite length or indefinite length arbitrary block data.

- If your instrument sends data in some other block data format, you can use the IOENTERB and IOOUTPUTB commands in conjunction with IOENTERS and IOOUTPUTS, respectively, to simulate these other formats.
- You should probably disable character matching before executing an IOENTERB or IOENTERAB because the character in the "match" string is generally a valid binary value, rather than a termination character.
- The commands that are sent to the scope are device dependent and are found in the manual for the scope.
- The error checking in the program consists of executing an IF statement after each call to an HP-IB command.
- Before an IOENTERS statement is executed, space should be allocated for the string by assigning the string to SPACE$(*n*) where *n* is the maximum length to be entered. If this is not done, the string will be of length 0, and no characters will be entered. You can then shorten the string to the correct length (in case less than the maximum number of characters were entered) by using the LEFT$ function.

The program has three main parts to it:

1. Read the data from the scope.
2. Print some statistics about the data.
3. Have the scope send the data to a printer.

```
999 rem this file should be appended to setup.bas or nodoc.bas
1050 'This program tells the scope to take a reading on channel 1, then
1060 'sends the data back to this program.  We can do anything we want
1070 'to the data at this time, and we choose to print some simple
1080 'statistics about the data.  The program then tells the scope to
1090 'send the data directly to the printer, illustrating how the
1100 'controller doesn't have to be directly involved in an HP-IB
1110 'transaction.
1440 '
1450 ISC = 7
1460 SCOPE = ISC * 100 + 7
1470 '
1480 'reset the HPIB interface
1490 '
```

```
1500 CALL IORESET(ISC)
1510 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1520 '
1530 'set up a timeout of 5 seconds
1540 '
1550 TIME = 5
1560 CALL IOTIMEOUT(ISC, TIME)
1570 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1580 '
1590 'clear the scope
1600 '
1610 CALL IOCLEAR(SCOPE)
1620 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1630 '
1640 'lockout the keyboard on the scope so the program has complete
1650 ' control of it
1660 '
1670 CALL IOREMOTE(ISC)
1680 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1690 '
1700 'set up some variables
1710 '
1730 DISABLE = 0 : ENABLE = 1
1740 EOL$ = CHR$(10)
1750 CRLF$ = CHR$(13) + CHR$(10)
1760 CMD$ = SPACE$(255)
1770 '
1780 '
1790 '
1800 'this is the beginning of part 1    ************************
1810 '     - get the data from the scope
1820 '
1830 '
1840 'setup scope to accept waveform data
1850 '
1860 CMD$ = "*RST"
1870 LENGTH = LEN(CMD$)
1880 CALL IOOUTPUTS(SCOPE, CMD$, LENGTH)
1890 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1900 '
1910 CMD$ = ":autoscale"
```

```
1920 LENGTH = LEN(CMD$)
1930 CALL IOOUTPUTS(SCOPE, CMD$, LENGTH)
1940 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1950 '
1960 'setup up the waveform source
1970 '
1980 CMD$ = ":waveform:format word"
1990 LENGTH = LEN(CMD$)
2000 CALL IOOUTPUTS(SCOPE, CMD$, LENGTH)
2010 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2020 '
2030 'input waveform preamble to controller
2040 '
2050 CMD$ = ":digitize channel1"
2060 LENGTH = LEN(CMD$)
2070 CALL IOOUTPUTS(SCOPE, CMD$, LENGTH)
2080 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2090 '
2100 CMD$ = ":waveform:preamble?"
2110 LENGTH = LEN(CMD$)
2120 CALL IOOUTPUTS(SCOPE, CMD$, LENGTH)
2130 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2140 '
2150 'read in the preamble
2160 '
2170 MAX = 10: ACTUAL = 0
2180 DIM PRE[10]
2190 CALL IOENTERA(SCOPE, PRE(0), MAX, ACTUAL)
2200 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2210 '
2220 'turn off 'lf' enter terminator, as 'lf' is a legal binary value
2230 '
2240 CALL IOMATCH(ISC, EOL$, DISABLE)
2250 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2260 '
2270 'make sure EOI is set
2280 '
2290 CALL IOEOI(ISC, ENABLE)
2300 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2310 '
2320 'command scope to send data.  It is sent in IEEE arbitrary block
```

Computer Museum

```
2330 'format.
2340 '
2350 CMD$ = ":waveform:data?"
2360 LENGTH = LEN(CMD$)
2370 CALL IOOUTPUTS(SCOPE, CMD$, LENGTH)
2380 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2390 '
2400 'enter the data
2410 '
2420 DIM READINGS%(5000): BYTES = 8000: ACTUAL = 0: FLAG = 2
2430 CALL IOENTERAB(SCOPE, READINGS%(0), BYTES, ACTUAL, FLAG)
2440 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2450 '
2460 'use IOGETTERM to see if we entered all the points
2470 '
2480 CALL IOGETTERM(ISC, REASON)
2490 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2500 IF (REASON AND 1) = 0 THEN PRINT "NOT ALL POINTS FOUND"
2510 '
2520 ' Read the last byte from the scope.  This must always be done
2530 ' after an IOENTERAB command.  If the character read is a
2540 ' 'lf', then the device is done sending data.  If the character
2550 ' read is a ';' or a ',', then the device is waiting to send
2560 ' another block of data.
2570 '
2580 ' Note also that we can use the select code instead of the device
2590 ' address for the first parameter of this command.  This is because
2600 ' the scope is still addressed to talk, and the computer to listen
2610 ' from the IOENTERAB command.
2620 '
2630 LENGTH = 1: ACTUAL = 0
2640 CMD$ = SPACE$(10)
2650 CALL IOENTERS (ISC, CMD$, LENGTH, ACTUAL)
2660 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2670 CMD$ = LEFT$(CMD$, ACTUAL)
2680 '
2690 IF CMD$ <> CHR$(10) THEN PRINT "scope wants to send more data..."
2700 '
2710 '
2720 'this is the beginning of part 2    **************************
2730 '     - print some statistics about the data
```

```
2740 '
2750 '
2760 'calculate minimum, maximum, and sum of the values in the data
2770 '
2780 VDIV   = 32 * PRE(8)
2790 OFFSET = (128 - PRE(10)) * PRE (8) + PRE (9)
2800 SDIV   = PRE(3) * PRE (5) / 10
2810 DELAY  = (PRE(3) / 2 - PRE(7)) * PRE(5) + PRE(6)
2820 '
2830 'Retrieve the scope's ID string
2840 '
2850 CMD$ = "*IDN?"
2860 LENGTH = LEN(CMD$)
2870 CALL IOOUTPUTS(SCOPE, CMD$, LENGTH)
2880 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2890 '
2900 LENGTH = 50: ACTUAL = 0
2910 CMD$ = SPACE$(50)
2920 CALL IOENTERS (SCOPE, CMD$, LENGTH, ACTUAL)
2930 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2940 '
2950 'print the stats
2960 '
2970 PRINT
2980 PRINT " Oscilloscope ID: ";CMD$
2990 PRINT " -------  statistics  --------"
3000 PRINT "      Volts/Div = "; VDIV;   " V"
3010 PRINT "         Offset = "; OFFSET; " V"
3020 PRINT "          S/Div = "; SDIV;   " S"
3030 PRINT "          Delay = "; DELAY;  " S"
3040 PRINT
3050 '
3060 '
3070 '
3080 'this is the beginning of part 3    ************************
3090 '     - have the scope send the graph directly to the printer
3100 '
3110 '
3120 'Next, let's tell the scope to print directly to a printer.
3130 ' We must first send the HP-IB commands to make the scope a talker
3140 ' and the printer a listener. This is done with the IOSEND command.
```

```
3150 ' We will tell the scope to issue a service request when it's done
3160 ' printing, as we need to wait for the printing to complete before
3170 ' continuing the program.
3180 '
3190 'tell the scope to SRQ on 'operation complete'
3200 '
3210 CMD$ = "*SRE 32 ; *ESE 1"
3220 LENGTH = LEN(CMD$)
3230 CALL IOOUTPUTS(SCOPE, CMD$, LENGTH)
3240 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
3250 '
3260 'tell scope to print
3270 '
3280 CMD$ = ":print? ; *OPC"
3290 LENGTH = LEN(CMD$)
3300 CALL IOOUTPUTS(SCOPE, CMD$, LENGTH)
3310 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
3320 '
3330 'tell scope to talk and printer to listen
3340 ' the listen command is formed by adding 32 to the device address
3350 '     of the device to be a listener
3360 ' the talk command is formed by adding 64 to the device address of
3370 '     the device to be a talker
3380 '
3390 PRINT.LISTEN$ = CHR$(32 + 1)   'printer is at device address 1
3400 SCOPE.TALK$   = CHR$(64 + 7)   'scope is at device address 7
3410 UNLISTEN$ = CHR$(63)
3420 '
3430 'send the command
3440 '
3450 CMD$ = UNLISTEN$ + PRINT.LISTEN$ + SCOPE.TALK$
3460 LENGTH = LEN(CMD$)
3470 CALL IOSEND(ISC, CMD$, LENGTH)
3480 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
3490 '
3500 'now, the ATN line must be set to FALSE.
3510 '
3520 COND = 8
3530 STATUS = 0
3540 CALL IOCONTROL(ISC, COND, STATUS)
3550 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

```
3560 '
3570 '
3580 'now wait for SRQ from scope
3590 '
3600 'NOT.DONE:
3610 '
3620 COND = 1: STATUS = 0
3630 CALL IOSTATUS(ISC, COND, STATUS)
3640 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
3650 '
3660 IF STATUS = 0 THEN GOTO 3600
3670 '
3680 'make sure it was the scope requesting service
3690 '
3700 CALL IOSPOLL(SCOPE, STATUS)
3710 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
3720 '
3730 ' 64 = bit 6 set
3740 '
3750 IF (STATUS AND 64) = 0 THEN GOTO 3600
3760 '
3770 ' Clear the status byte so the scope can assert SRQ again
3780 ' if needed
3790 '
3800 CMD$ = "*CLS"
3810 LENGTH = LEN(CMD$)
3820 CALL IOOUTPUTS(SCOPE, CMD$, LENGTH)
3830 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
3840 '
3850 'give local control back to the scope
3860 '
3870 CALL IOLOCAL(ISC)
3880 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
3890 '
3900 SYSTEM
```

Multimeter Example

```
998  ' This program should be appended to SETUP.BAS or NODOC.BAS
999  '
1000 ' This example uses the HP 34401A Multimeter as the primary device.
1010 ' We will also use the HP 3325A Function Generator as a source for
1020 ' the multimeter.
1030 '
1040 ' This example sets up the meter to take 128 readings, reads the data
1050 ' into an array, then plots the data on the screen.  In effect, it
1060 ' turns the multimeter into a simple oscilloscope.  This program is
1070 ' also checking other devices that are on the bus to see if they need
1080 ' service.  The SRQ line along with parallel and serial polling is
1090 ' used to make these checks.  The program will continue until the user
1100 ' presses the F1 key on the PC keyboard.
1110 '
1380   '
1390   NUM.READINGS = 128
1400   ISC = 7                'interface select code
1410   SOURCE = ISC * 100+12  'address of the function generator
1420   DVM = ISC * 100+22     'address of the digital volt meter
1430   DIM READINGS![128]     'place we will put the readings from the dvm
1440   '
1450   'the F1 key will end the program
1460   '
1470   ON KEY(1) GOSUB 3730 'QUIT
1480   KEY(1) ON
1490   '
1500   'reset the HPIB interface
1510   '
1520   CALL IORESET(ISC)
1530   IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1540   '
1550   'set up a timeout of 5 seconds
1560   '
1570   TIME = 5
1580   CALL IOTIMEOUT(ISC, TIME)
1590   IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1600   '
1610   'clear the devices we are going to use
```

```
1620   '
1630   CALL IOCLEAR(SOURCE)
1640   IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1650   '
1660   CALL IOCLEAR(DVM)
1670   IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1680   '
1690   'we meet the minimum requirements to use IOFASTOUT, so let's use it
1700   '
1710   CALL IOFASTOUT(ISC, TRUE)
1720   IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1730   '
1740   NULL$ = ""
1750   CALL IOEOL(ISC, NULL$, FALSE)
1760   IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1770   '
1780   'We will now configure all devices that can respond to a parallel
1790   ' poll.  This example assumes devices at addresses 20 and 7 can
1800   ' respond to a parallel poll.  See operators manual of individual
1810   ' devices to see if they can respond to a parallel poll.
1820   '
1830   'configure the device at address 20 for a parallel poll
1840   '
1850   DEVICE.ADDR = ISC * 100 + 20
1860   CONFIGURATION = &H08 'RESPOND WITH A "1" ON LINE 0
1870   CALL IOPPOLLC(DEVICE.ADDR, CONFIGURATION)
1880   IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1890   '
1900   'configure the device at address 7 for a parallel poll
1910   '
1920   DEVICE.ADDR = ISC * 100 + 7
1930   CONFIGURATION = &H09 'RESPOND WITH A "1" ON LINE 1
1940   CALL IOPPOLLC(DEVICE.ADDR, CONFIGURATION)
1950   IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
1960   '
1970   'configure any other devices that can respond to parallel poll here
1980   '
1990   '
2000   'let's use dma to send the strings to program the devices
2010   '
2020   COUNT = 40: CHANNEL = 3
```

```
2030    CALL IODMA(ISC, COUNT, CHANNEL)
2040    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2050    '
2060    'program the function generator
2070    '
2080    CODES$ = "RF1 FR30HZ FU1 ST1KH SP10KH MF1KH AM1VR TI5SE"
2090    LENGTH = LEN(CODES$)
2100    CALL IOOUTPUTS(SOURCE, CODES$, LENGTH)
2110    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2120    '
2130    'program the dvm
2140    '
2150    CODES$ = ":CONF:VOLT:DC 30,.1;"
2160    CODES$ = CODES$ + ":ZERO:AUTO OFF;"
2170    CODES$ = CODES$ + ":TRIG:DELAY MIN;"
2180    CODES$ = CODES$ + ":DISP:STATE OFF;"
2190    LENGTH = LEN(CODES$)
2200    CALL IOOUTPUTS (DVM, CODES$, LENGTH)
2210    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2220    '
2230    'turn dma off again
2240    '
2250    COUNT = 0: CHANNEL = 3
2260    CALL IODMA(ISC, COUNT, CHANNEL)
2270    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2280    '
2290    'turn on automatic srq checking
2300    '
2310    ON PEN GOSUB 2530 'SRQ.HANDLER
2320    PEN ON
2330    '
2340    PRIORITY = 0
2350    CALL IOPEN(ISC, PRIORITY)
2360    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2370    '
2380    '
2390    'BACK:
2400        '
2410        'can have controller do other work here
2420        '
2430        GOSUB 3170 'GET.DATA
```

```
2440        '
2450        GOTO 2390 'BACK
2460    '
2470    'end of main program.  support routines follow.
2480    '
2490    '
2500    'this is the routine defined by the 'on pen gosub' statement above.
2510    ' this routine will be called each time an srq comes in.
2520    '
2530    'SRQ.HANDLER:
2540        '
2550        'conduct a parallel poll
2560        'note that the source doesn't respond to parallel poll's,
2570        ' so we need to poll that device separately.
2580        '
2590        RESPONSE = 0
2600        CALL IOPPOLL(ISC, RESPONSE)
2610        IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2620        IF (RESPONSE AND 1) THEN GOSUB 2850 'POLL.DEVICE.1
2630        IF (RESPONSE AND 2) THEN GOSUB 3010 'POLL.DEVICE.2
2640        '
2650        'check all devices that were configured to respond to
2660        ' parallel poll
2670        '
2680        'check any other devices on the bus here that weren't
2690        ' configured to respond to parallel poll by performing
2700        ' a serial poll on each one.
2710        '
2720        CALL IOSPOLL(SOURCE, RESPONSE)
2730        IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2740        '
2750        'see if we've cleared the srq yet
2760        '
2770        STAT = 1: RESPONSE = 0
2780        CALL IOSTATUS(ISC, STAT, RESPONSE)
2790        IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2800        IF RESPONSE=1 THEN PRINT "SRQ LOCKED HIGH": GOTO 3730 'QUIT
2810        '
2820    RETURN
2830    '
2840    '
```

```
2850   'POLL.DEVICE.1:
2860       '
2870       'do a serial poll of the device configured to use parallel
2880       ' poll line 0
2890       '
2900       DEVICE.ADDR = ISC * 100 + 20
2910       CALL IOSPOLL(DEVICE.ADDR, RESPONSE)
2920       IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
2930       '
2940       'should check RESPONSE here to see if any action needs to be
2950       ' taken.  The values that RESPONSE can take are device
2960       ' dependent.
2970       '
2980       RETURN
2990   '
3000   '
3010   'POLL.DEVICE.2:
3020       '
3030       'do a serial poll of the device configured to use parallel
3040       ' poll line 1
3050       '
3060       DEVICE.ADDR = ISC * 100 + 7
3070       CALL IOSPOLL(DEVICE.ADDR, RESPONSE)
3080       IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
3090       '
3100       'should check RESPONSE here to see if any action needs to be
3110       ' taken.  The values that RESPONSE can take are device
3120       ' dependent.
3130       '
3140       RETURN
3150   '
3160   '
3170   'GET.DATA:
3180       '
3190       'Ask the DVM to send us the data
3200       '
3210       CODES$ = ":SAMPLE:COUNT 128;"
3220       CODES$ = CODES$ + ":READ?"
3230       LENGTH = LEN(CODES$)
3240       CALL IOOUTPUTS (DVM, CODES$, LENGTH)
3250       IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

```
3260        '
3270        'Read in the data
3280        '
3290        LENGTH = NUM.READINGS
3300        ACTUAL = 0
3310        CALL IOENTERA (DVM, READINGS! (0), LENGTH, ACTUAL)
3320        IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
3330        '
3340        'set graphics mode and draw border on screen
3350        '
3360        CLS: SCREEN 2
3370        WINDOW (0,0)-(639,199)
3380        LINE (0,0)-(639,199),,B
3390        '
3400        'calculate min and max values for y-axis
3410        '
3420        YMAX = READINGS![0] : YMIN = READINGS![0]
3430        FOR I = 1 TO ACTUAL
3440            IF READINGS![I] < YMIN THEN YMIN = READINGS![I]
3450            IF READINGS![I] > YMAX THEN YMAX = READINGS![I]
3460        NEXT I
3470        '
3480        'print graph labels
3490        '
3500        LOCATE 2,2 : PRINT "MAX = ";YMAX;
3510        LOCATE 24,2 : PRINT "MIN = ";YMIN;
3520        '
3530        'scale ymin and ymax so there is space between graph and border
3540        '
3550        IF YMIN > 0 THEN YMIN = YMIN*0.6 ELSE YMIN = YMIN*1.4
3560        IF YMAX > 0 THEN YMAX = YMAX*1.4 ELSE YMAX = YMAX*0.6
3570        '
3580        'graph the data
3590        '
3600        WINDOW (0,YMIN)-(ACTUAL-1,YMAX)
3610        XAXIS = 1
3620        PREV = READINGS![XAXIS-1]
3630        PSET(XAXIS-1, PREV)
3640        WHILE (XAXIS < ACTUAL)
3650          CURRENT = READINGS![XAXIS]
3660          LINE (XAXIS-1, PREV)-(XAXIS,CURRENT)
```

```
3670        PREV = CURRENT
3680        XAXIS = XAXIS + 1
3690      WEND
3700      RETURN
3710  '
3720  '
3730  'QUIT:
3740      '
3750      'clear the dvm so we can send the commands to reset it
3760      '
3770      CALL IOCLEAR(DVM)
3780      IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
3790      '
3800      'reset the dvm
3810      '
3820      CODES$ = ":DISPLAY:STATE ON; *RST"
3830      LENGTH = LEN(CODES$)
3840      CALL IOOUTPUTS (DVM, CODES$, LENGTH)
3850      IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
3860      '
3870      'unconfigure the parallel poll
3880      '
3890      CALL IOPPOLLU(ISC)
3900      IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
3910      '
3920      SYSTEM
```

3

QuickBASIC and QBasic Programming

Introduction

This chapter explains how to use the HP-IB Command Library with QuickBASIC, QBasic, and similar BASIC programming languages. Unless noted otherwise, all references in this chapter to QuickBASIC apply to all similar BASIC languages including QBasic.

Supported versions of QuickBASIC and similar languages are listed on the *Supported Languages* sheet included with the Command Library. For example, you can use certain versions of Microsoft QuickBASIC and the Microsoft BASIC Compiler.

This chapter contains several sections describing how you can use the Command Library with QuickBASIC:

- Copying the necessary Library files to a work disk.
- Creating, compiling, and running a QuickBASIC program.
- Processing errors.
- Learning about parameters for Library commands.
- Checking example programs. Two listings at the end of this chapter show how you can use Library commands in QuickBASIC programs.

Detailed syntax information for the commands as they're used with QuickBASIC is included in chapter 4, "BASIC Reference."

Copying Files

To begin programming in BASIC, you must copy the BASIC Library files to your work disk.

The HP-IB Command Library disks contain an INSTALL program that copies the QuickBASIC Library files to your system for you.

To use INSTALL:

1. Insert the Library disk into your flexible disk drive—if you're using 5.25-inch disks, use the disk labeled "Disk 1—Install."
2. Run INSTALL by typing

   ```
   a:install (Enter)
   ```

3. Follow the instructions displayed on the screen. When you successfully complete the instructions, the following files are copied:

   - **For QuickBASIC 4.0 or later:**

     QBHPIB.LIB
     QBHPIB.QLB
     QBSETUP.BAS
     QEXAMPLE.BAS
     QBCL.BAT
     QBSCOPE.BAS
     QBMETER.BAS

   - **For Compiled BASIC:**

     QBHPIB.LIB
     QBSETUP.BAS
     QEXAMPLE.BAS
     CBCL.BAT
     QBSCOPE.BAS
     QBMETER.BAS

   - **For QBasic:**

     QHPIB.LIB
     QSETUP.BAS
     QCL.BAT

- **For Microsoft BASIC 7.0 and 7.1 Professional Development System:**

  QBXHPIB.LIB
  QBXHPIB.QLB
  QBSETUP.BAS
  QBXCL.BAT
  QEXAMPLE.BAS
  QBSCOPE.BAS
  QBMETER.BAS

If you specified invalid drives, or if the system disk is write-protected, no files will be copied. Also make sure you copy all the necessary files from your BASIC compiler and linker to your system. Refer to your BASIC manual for details.

Programming in QuickBASIC

You can create a QuickBASIC program using a text editor, the QuickBASIC integrated environment, or the interpreted BASIC environment.

If you use the QuickBASIC interactive environment, you will need to load the HP-IB Quick library with it by invoking QuickBASIC as:

```
QB /L QBHPIB
```

This incorporates the HP-IB library into the interactive environment so the HP-IB routines will work.

The following diagram shows the general structure of your application program.

```
User-defined COMMON declarations
.
.

REM $INCLUDE: 'QBSETUP.BAS'

Your BASIC application program
.
.
```

If you will be chaining programs, decide if you need any COMMON declarations for your application. These must appear before any executable statements.

User-defined COMMON statements should be followed by a metacommand to include the QBSETUP.BAS file. For QuickBASIC programming, the Library is implemented as a set of assembly language subroutine calls. QBSETUP.BAS is a header for your application program. It sets up the necessary error status variables, declares some working variables, contains related COMMON statements and establishes an error handling routine.

Your application programs should start after the statement to include QBSETUP.BAS.

If you are using multiple modules in QuickBASIC or Compiled BASIC, we recommend editing the file QBSETUP.BAS to change all COMMON statements to COMMON SHARED. Then add these COMMON SHARED statements to the beginning of each module. Refer to your BASIC manual for more details on how to use the COMMON statement.

Dimensioning Arrays

While the syntax of QuickBASIC allows dynamic arrays, they are not supported by the HP-IB Command Library. This applies to arrays declared with the $DYNAMIC attribute as well as arrays declared using variables for the dimensions. For example:

Correct	`DIM READINGS(20)`

Incorrect	`MAX.ELEMENTS% = 20`
	`DIM READINGS(MAX.ELEMENTS%)`

Chaining in BASIC

Every chained-to program must include several statements from QBSETUP.BAS.

- Include the three COMMON declarations from QBSETUP.BAS:

  ```
  COMMON PCIB.BASERR,PCIB.ERR,PCIB.ERR$,PCIB.NAME$,PCIB.GLBERR
  COMMON FALSE%,TRUE%,NOERR,EUNKNOWN,ESEL,ERANGE,ETIME,ECTRL,EPASS
  COMMON ENUM,EADDR
  ```

  These lines should follow your own COMMON statements, and all COMMON statements must appear before any executable statements.

  The order in which COMMON variables are listed is significant. Mismatched COMMONs between chained programs can cause execution errors.

- Include the call to DEFERR as shown in QBSETUP.BAS:

  ```
  CALL DEFERR (PCIB.ERR,PCIB.ERR$,PCIB.NAME$,PCIB.GLBERR)
  ```

  DEFERR tells the HP-IB Library where in memory your error variables (such as PCIB.ERR) are located.

If you have sufficient space in your chained-to program, you can simply include the entire QBSETUP.BAS file (following any user-defined COMMON statements). Although only the COMMON statements and DEFERR call are essential for chaining, QBSETUP.BAS also provides an error handling routine, initializes the error variables, and sets up the error mnemonics (such as ESEL).

The behavior of QuickBASIC programs that use chaining depends on whether they are linked with the BASIC "run" library or with the "compile" library. For example, due to a limitation in QuickBASIC, the values of COMMON variables are not preserved across a chain if the program was compiled to run as a stand-alone program. To work around this limitation, you should compile your program to require the file BRUN4x.EXE. To do this, simply add a /E switch to the DOS command line when compiling. For example, here is how to compile the program 'HELLO.BAS':

```
BC HELLO /E/V/W;
LINK HELLO, HELLO,,QBHPIB
```

Computer Museum

You can also compile from within the environment and create an executable that requires BRUN4x.EXE.

If you will be chaining programs, read those portions of your BASIC manual relating to compiling, linking, and running programs before you link a program.

Writing a BASIC Program

In an application program, you typically use the Library commands in the following manner to execute an operation:

1. Set up the required variables.
2. Perform the operation.
3. Test to see if the operation completed successfully.

In this example, you follow these steps to program two instruments—an HP 3325A Synthesizer/Function Generator and an HP 3456A Digital Voltmeter. You program the source to output a 2-V rms signal, swept from 1 kHz to 10 kHz. You program the DVM to take 20 readings from the signal and output them to an array. Finally, you display the readings on the screen.

Use a convenient editor and begin with the $INCLUDE metacommand:

```
REM $INCLUDE: 'QBSETUP'
```

Be sure that you leave *no spaces* between the $ and INCLUDE.

Since this example does not chain programs, no extra COMMON statements are needed.

1. Define main program with working variables.

```
REM $INCLUDE: 'QBSETUP'

OPTION BASE 1
MAX.ELEMENTS% = 20
DIM READINGS (20)
ACT.ELEMENTS% = 0
CALL INITIALIZE (ISC&, DVM&, SOURCE&)
CALL SOURCESETUP (SOURCE&)
CALL DVMSETUP (DVM&)
CALL TRIGGER (DVM&, SOURCE&)
CALL WAITFORSRQ (ISC&, DVM&)
CALL TAKEREADINGS (ISC&, DVM&, MAX.ELEMENTS%, ACT.ELEMENTS%)
CALL PRINTREADINGS (ACT.ELEMENTS%)
END
```

- Set the array base to 1 so array element numbering begins with 1 instead of 0.
- Define a maximum-readings variable (MAX.ELEMENTS).
- Dimension an array (READINGS) to hold the readings taken by the voltmeter. This must be dimensioned using a literal, not a variable.
- Set ACT.ELEMENTS, the actual number of elements read by IOENTERA, to 0.
- Each subprogram is described as it is presented below.

2. Write the initialization subprogram.

```
SUB INITIALIZE (ISC, DVM, SOURCE)
SHARED PCIB.ERR, PCIB.BASERR, NOERR

  ISC% = 7
  DVM% = 722
  SOURCE% = 717
  CALL IORESET (ISC%)
  IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
  TIMEOUT! = 5.0
  CALL IOTIMEOUT (ISC%, TIMEOUT!)
  IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
  CALL IOCLEAR (ISC%)
  IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
END SUB
```

- Set the interface select code variable ISC to 7.
- Set the voltmeter address variable DVM to 722.
- Set the signal source address variable SOURCE to 717. This example assumes select code 7, voltmeter address 22, and source address 17.
- Set the interface to its default configuration.
- Define a system timeout of 5 seconds.
- Perform IOCLEAR to put all instruments into a known, device-dependent state.

Note

This program includes an error checking line after each Library command:

```
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

If there is an error (that is, the error variable PCIB.ERR does not equal NOERR), this line calls the QBSETUP.BAS error handling routine and passes the error number PCIB.BASERR. The routine then prints error information and stops the program. See "Processing I/O Errors" in chapter 1 and "BASIC Error Handling" later in this chapter for more information.

3. Generate a subprogram to set up the source.

```
SUB SOURCESETUP (SOURCE&)
SHARED PCIB.ERR, PCIB.BASERR, NOERR

  CODES$ = "RF2 FU1 ST1KH SP10KH MF1KH AM2VR TI5SE"
  LENGTH% = LEN(CODES$)
  CALL IOOUTPUTS (SOURCE&,CODES$,LENGTH%)
  IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
END SUB
```

- Define the programming codes string CODES$ to hold:

  RF2—select the rear panel signal output.
  FU1—select the sine wave function.
  ST1KH—select a starting frequency of 1 kHz.
  ST10KH—select a stopping frequency of 10 kHz.
  MF1KH—select a marker frequency of 1 kHz.
  AM2VR—select an amplitude of 2 V rms.
  TI5SE—select a sweep time of 5 seconds.

- Use the IOOUTPUTS command to send the programming codes to the source with the proper length parameter.

4. Generate a subprogram to set up the voltmeter.

```
SUB DVMSETUP (DVM&)
SHARED PCIB.ERR, PCIB.BASERR, NOERR

  CODES$ = "H SM004 F2 R4 FL0 Z0 4STG 20STN RS1 T4"
  LENGTH% = LEN(CODES$)
  CALL IOOUTPUTS (DVM&,CODES$,LENGTH%)
  IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
END SUB
```

- Define the programming codes string CODES$ to hold:

  H—software-reset the voltmeter.

  SM004—set the service request mask to enable the voltmeter to set the interface SRQ line when it finishes taking readings (when the Data Ready bit of the serial poll response byte is set).

  F2—select the AC volts function.
  R4—select the 10 volt range.
  FL0—turn off filtering.
  Z0—turn off auto zero.
  4STG—select the 4-digit display.
  20STN—take 20 readings.
  RS1—turn on reading storage.
  T4—select trigger hold.

- Send the programming codes to the voltmeter with the proper length parameter.

5. Define a subprogram to trigger the instruments.

```
SUB TRIGGER (DVM&, SOURCE&)
SHARED PCIB.ERR, PCIB.BASERR, NOERR

  CALL IOTRIGGER(DVM&)

  IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
  CODES$ = "SS"
  LENGTH% = LEN(CODES$)
  CALL IOOUTPUTS (SOURCE&,CODES$,LENGTH%)
  IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
END SUB
```

- Execute the IOTRIGGER command for the voltmeter.
- Define the programming codes string to hold the source's trigger code.
- Send the programming codes to the source with the proper length parameter.

These lines demonstrate that some instruments respond to an HP-IB trigger command, while others must be triggered with instrument-specific programming codes.

6. Wait for the voltmeter to finish reading.

```
SUB WAITFORSRQ (ISC&, DVM&)
SHARED PCIB.ERR, PCIB.BASERR, NOERR

           SRQ% = 1
CHECKSTAT: CALL IOSTATUS (ISC&, SRQ%, STATUS%)
           IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
           IF STATUS% = 0 THEN GOTO CHECKSTAT
           CALL IOSPOLL (DVM&, RESPONSE%)
           IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
           IF (RESPONSE% AND 68) <> 68 THEN GOTO CHECKSTAT
END SUB
```

- Define the interface condition whose status is being checked. In this case, check for condition 1—is the SRQ line set?

- Execute the status command and return the result in STATUS.
- As long as STATUS is 0, the SRQ line is not set, indicating that the voltmeter is not finished taking readings. As soon as it changes to 1, perform a serial poll on the voltmeter to learn which of its conditions, if any, set the SRQ line. The serial poll also clears the request.
- The result of the serial poll is the status byte of the voltmeter, returned in RESPONSE. Compare RESPONSE and the value 68—the sum of the Request Service bit (64) and the Data Ready bit (4). If these bits are set, continue because the voltmeter is finished. If they are not set, perform the status check again.
- Note that alphanumeric labels (such as CHECKSTAT) may be used.

7. Enter the readings into an array.

```
SUB TAKEREADINGS (ISC&,DVM&,MAX.ELEMENTS%,ACT.ELEMENTS%)
SHARED PCIB.ERR, PCIB.BASERR, NOERR, READINGS()

  CODES$ = "S01 -20STR RER"
  LENGTH% = LEN(CODES$)
  CALL IOOUTPUTS (DVM&,CODES$,LENGTH%)
  IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
  '
  STATE% = 0
  CALL IOEOI (ISC&,STATE%)
  IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
  CALL IOENTERA(DVM&,SEG READINGS(1),MAX.ELEMENTS%,ACT.ELEMENTS%)
  IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
END SUB
```

- Define the programming codes string to direct the voltmeter to output its stored readings:

  SO1—turn on system output mode.
  −20STR—unstore the 20 readings from register R.
  RER—recall (readout) the 20 readings.

- Output the programming codes to the voltmeter with the proper length parameter.

- Disable the EOI mode so reading won't terminate after entering only one value.
- Enter the voltmeter readings into the READINGS array starting at the first element. Include the maximum and actual length parameters. READINGS(1) here refers to the first element of READINGS.

8. Print the readings.

```
SUB PRINTREADINGS (ACTUAL%)
SHARED READINGS()

  PRINT "THE READINGS ARE: "
  FOR I% = 1 TO ACTUAL%
    PRINT I%, READINGS(I)
  NEXT I%
END SUB
```

Saving the BASIC Program

When you have finished typing your program, check it for typographical errors. Then exit your text editor, saving your program with the name QBPROG.BAS. If you're using a general text editor or the interpreted BASIC editor, be sure to save it in ASCII format (most text editors do so by default).

Note

If you are using the interpreted BASIC environment to write your program, you must use the A option to save it:

```
save "qbprog",a Enter
```

This saves your program in ASCII format and appends the suffix ".BAS".

Compiling and Linking the BASIC Program

QuickBASIC provides three methods for compiling and linking a BASIC program:

- Compiling the program with the /O option, linking the new object file with the alternate run-time library and the Command Library, and running the new executable file.
- Compiling the program without the /O option, linking the new object file with the run-time-module library and the Command Library, and running the new executable file with the run-time module present.
- Specifying the library when you start the QuickBASIC integrated environment, then running the program from there.

The method used affects such factors as the size of the executable code generated, execution speed of your program, and the meaning of any CHAIN and COMMON statements in your program.

This manual describes use of the HP-IB Command Library with the first and last methods. For further information, read those portions of your BASIC manual relating to compiling, linking, and running BASIC programs.

Automatic Compiling and Linking

A batch file in your Command Library automatically compiles and links a specified BASIC program. The file you use depends upon your BASIC language.

If you use this method, you should make sure your PATH environment variable includes the Command Library directory and the BASIC executables directory. For example, the following command in your AUTOEXEC.BAT file includes the HPIB and BAS directories in the PATH variable

```
path=c:\;c:\dos;c:\bas;c:\hpib
```

To compile and link a program, type the following at the MS-DOS prompt, where *progname* is the name of a BASIC program (without the .BAS extension appended).

- QuickBASIC 4.0 or later (uses file QBCL.BAT)

  `qbcl` *progname* Enter

- Microsoft Compiled BASIC (uses file CBCL.BAT)

  `cbcl` *progname* (Enter)

- Microsoft BASIC PDS (uses file QBXCL.BAT)

  `qbxcl` *progname* (Enter)

Note

If your program includes any ON-event branching (such as ON PEN for service requests), you should edit the .BAT file named above to include the /V and /W switches in its commands. This enables end-of-command event checking.

The original .BAT file doesn't use these switches, but they're present in the file as unexecuted lines.

For example, to compile and link the program described above using QuickBASIC 4.0, type

`qbcl qbprog` (Enter)

Compiling and Linking Separately

To only compile a program, type the compiler command followed by the file name followed by the /E and /O switches and any other compiler switches you need (see the note below).

For QuickBASIC 4.0 or later, Compiled BASIC, or BASIC PDS:

`bc` *progname* `/e /o;` (Enter)

The compiler requires the /E switch to process the error routines since QBSETUP.BAS uses an ON ERROR branch. The /O switch directs the compiler to assume it will use the alternate run-time library instead of the run-time-module library.

Note

If your program includes any ON-event branching (such as ON PEN for service requests), you should also include the /V and /W switches to enable end-of-command event checking.

If your program *doesn't* use ON-event branching, omit the /V and /W switches to improve performance.

For example, to compile the example program

```
bc qbprog /e /o; [Enter]
```

This command generates an object listing named QBPROG.OBJ.

The object file produced by the compiler from your program must be linked with the QuickBASIC HP-IB Library (QBHPIB.LIB) before you run your program. The linker matches up your HP-IB Library calls with the proper subroutines from the library. Any compiled program must be linked prior to execution, even if it does not use the HP-IB Library.

Note

Be sure to use the linker distributed with your compiler. Differences between versions of LINK.EXE can result in improper linking if you mix compilers and linkers.

To link your program and libraries, type

```
link [Enter]
```

As linking executes, several prompts appear on the screen:

1. When you are prompted for object modules, type

   ```
   qbprog [Enter]
   ```

   The linker appends the extension .OBJ to the filename and then searches for QBPROG.OBJ.
2. When you are prompted for a run file, press [Enter]. The default QBPROG.EXE is assumed.
3. When you are prompted for a list file, press [Enter]. A listing file is not required for this example.

4. When you are prompted for runtime libraries, type your Command Library name:

   - QuickBASIC 4.0 and later: QBHPIB.
   - Microsoft Compiled BASIC: QBHPIB.
   - Microsoft PDS Compiler: QBXHPIB.

   For example,

   ```
   qbhpib (Enter)
   ```

   This directs the linker to link the Command Library to your program. The linker also automatically links the alternative run-time library.

When the linker returns with the operating system prompt, you have an executable file named QBPROG.EXE that is ready to run.

See your reference manual for more information on the compiler and linker.

Compiling and Linking in the QuickBASIC Environment

To compile, link, and run a program from the QuickBASIC integrated environment, you should start QuickBASIC 4.0 using the following command (and the QBHPIB.QLB library):

`qb` *progname* `/l qbhpib` (Enter)

This loads the Command Library into the QuickBASIC environment so you can run your program. See your QuickBASIC manual for more information about the environment.

Running the BASIC Program

When you are ready to run the program, connect the SIGNAL output of the source to the VOLTS input of the DVM. (Include a 50-ohm load in this line to ensure proper readings.) Use HP-IB cables to connect the instruments to your computer.

To execute your program, type the name of the .EXE file—in this case QBPROG

```
qbprog (Enter)
```

Watch the display on the function generator. You will see the various functions (sine wave, AC volts, sweep time) displayed as they are programmed.

The voltmeter displays its operation as well—you can watch it take readings, store them, and output them to the READINGS array. As the program ends, it displays the readings on your screen.

Note

If the name of a Library command is misspelled, or if the number or types of parameters is wrong, system restart may be required to recover.

BASIC Error Handling

General information about Command Library errors, BASIC error variables, and how to process errors are contained in "Processing I/O Errors" in chapter 1 and in appendix A, "Error Descriptions."

If you don't need to write a special error-processing routine for your program, you can use the one provided in QBSETUP.BAS.

```
' Error handling routine
'
ERRORHANDLER:
99  IF ERR = PCIB.BASERR THEN GOTO LIBERROR
    PRINT "BASIC error #"; ERR; "occurred on line #"; ERL
    STOP

LIBERROR:
  TMPERR = PCIB.ERR
  IF TMPERR = 0 THEN TMPERR = PCIB.GLBERR
  PRINT "HP-IB error #"; TMPERR; " detected on line #"; ERL
  PRINT "Error: "; PCIB.ERR$
  STOP
```

This error handling routine is based on the BASIC function ERROR and its associated variables ERR and ERL. (If you need more details about ERROR, ERR and ERL, refer to your BASIC manual.) The variable PCIB.BASERR (value 255) provides a mechanism for differentiating between a BASIC error and an HP-IB Command Library error. If ERR = PCIB.BASERR, the error is generated by the Command Library—otherwise, it's a BASIC system error.

The above routine prints the type of error (BASIC or HP-IB), the error number, the error message, and the line on which the error was detected. Then, it stops the program. An ON ERROR statement in the QBSETUP.BAS file defines a branch to the error handling routine for any BASIC or program-initiated call to the ERROR function.

Note

If your program has unnumbered lines, the indicated line number is the last numbered line encountered—most likely line 99 in QBSETUP.BAS.

Programming in QBasic

QBasic programming is very similar to QuickBASIC programming with these exceptions:

- You must include the file `QSETUP.BAS` within all QBasic programs. You can do this using a DOS COPY command:

  ```
  COPY QSETUP.BAS+MYPROG.BAS FINAL.BAS
  ```

  At this point, you run the program `FINAL` in the QBasic environment.

- These commands are *not* supported in QBasic:

  IOPASSCTL

  IOTAKECTL

  IOREQUEST

  IOPEN

  If you need to use any of these commands, you must use QuickBASIC instead of QBasic.

- These commands have a slightly different syntax in QuickBASIC and QBasic:

  IOENTERA

  IOENTERAB

  IOENTERB

  IOOUTPUTA

  IOOUTPUTAB

  IOOUTPUTB

  Refer to Chapter 4 for details about the syntax of these commands.

Command Library Parameters

This section presents information about Command Library parameters as they are used with QuickBASIC.

Note that if you are using any DEF statement (such as DEFINT) in QuickBASIC or compiled BASIC, you must make all HP-IB variables become real numbers by using the ! suffix (for example, ERANGE!).

Passing Parameters

In QuickBASIC, all parameters used in CALL statements can be passed by reference. That is, you can use variable names as parameters. However, for QuickBASIC 4.0 and later and Microsoft Compiled BASIC, you can also use literals and expressions for simple parameters that provide information to the command—but *not* for parameters that return information.

For example, the IOSPOLL command uses two parameters: a device address and a response value. The address provides information, so it can be a variable, literal, or expression. The response returns information, so it must be a variable.

Variables are always valid as parameters. For example

```
ISC& = 7
CALL IOSPOLL (ISC&,RESPONSE%)
```

Parameter Types

Several types of variables are used to describe parameters to Library command calls in chapter 4, "BASIC Reference."

Integer Variable

An integer variable is distinguished from other identifiers by "%" appended to the variable name. The valid range for this variable type is integer numbers from −32,768 to 32,767. Integers are used to specify flags and other discrete information. For example:

```
FLAG%
STATUS%
```

Long-Integer Variable

A long-integer variable is distinguished from other identifiers by "&" appended to the variable name. The valid range for this variable type is integer numbers from −2,147,483,648 to 2,147,483,647. Long-integer variables are used to specify select codes and device addresses, including extended addresses. For example:

```
ISCDEVICE.ADDRESS&
```

Single-Precision Real Variable

A single-precision real variable is distinguished from other identifiers either by "!" appended to the variable name, or by no suffix appended at all. The valid range for these variables is approximately 2×10^{-39} to $2 \times 10^{+38}$ (negative or positive). Single-precision real numbers have approximately seven digits of accuracy. They're used to specify numeric data. For example:

```
VALUE
READING!
```

Single-Precision Real Array

A single-precision real array is an array of single-precision real numbers. Arrays are declared using the DIM statement. Although the theoretical maximum size for a BASIC array is 32,767, the actual limit is approximately 14,000 elements depending on the length of your program.

The Command Library supports only STATIC arrays. Dynamic arrays *must not* be used as parameters to Library calls.

BASIC does not permit the use of unsubscripted array names as parameters to CALL statements. Therefore, when you use arrays in Library function calls, use the first element of the array to be accessed as the parameter. In addition, the SEG keyword must be included with the array name. For example:

`SEG VALUES(0)` *Starts at element 0.*
`SEG VALUES(5)` *Starts at element 5.*

Be sure the number-of-elements parameter value does not exceed the number of elements available in the array. For data output, the Library continues through memory sending the contents of the individual cells until the specified count is satisfied. For data input, the Library continues through memory, writing over existing data or programs.

String Variable

A string variable is a variable that can contain any valid sequence of 0 or more ASCII characters. It is identified by a "$" appended to its identifier. The length of a string variable is not fixed, but may be anywhere from 0 (the null string) to 32,767 theoretically. However, the actual upper limit ranges from about 19,000 to 30,000 depending upon your compiler and the amount of code and data in your program. String variables are used to specify characters and other ASCII text. For example

```
COMMANDS$
```

String variables should be initialized with SPACE$ before data is entered with an IOENTERS command. Otherwise, if the current length is zero, no data will be transferred.

Any Type of Array

Any type of array can be a single-precision real array, an integer array, a double-precision real array, or other type of numeric array. (The array can't be a string.) It indicates the place to start reading or storing data.

The Command Library supports only STATIC arrays. Dynamic arrays *must not* be used as parameters to Library calls.

BASIC does not permit the use of unsubscripted array names as parameters to CALL statements. Therefore, when you use arrays in Library function calls, use the first element of the array to be accessed as the parameter. In addition, the SEG keyword must be included with the array name. For example:

`SEG VALUES(0)`	*Single-precision real; starts at first element.*
`SEG DATA#(5)`	*Double-precision real; starts at element 5.*
`SEG FLAGS%(0)`	*Integer; starts at first element.*

Be sure the number-of-bytes parameter value does not exceed the number of bytes available in the array—the number of elements times the number of bytes per element. For data output, the Library continues through memory sending the contents of the individual cells until the specified count is satisfied. For data input, the Library continues through memory, writing over existing data or programs.

Example Programs

Oscilloscope Example

The following program is written in QuickBASIC 4.0. The program uses two devices: an HP 54601A digitizing oscilloscope (or compatible scope) and a printer capable of printing HP Raster Graphics Standard, such as a ThinkJet printer.

The program tells the scope to take a reading on channel 1 and send the data back to this program. Then it prints some simple statistics about the data. 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 HP-IB transaction.

Things to note about this program:

- Note the use of the IOENTERAB command. This command will read an arbitrary block of data as defined in IEEE-488.2. IOENTERAB can read either definite length or indefinite length arbitrary block data.
- If your instrument sends data in some other block data format, you can use the IOENTERB and IOOUTPUTB commands in conjunction with IOENTERS and IOOUTPUTS, respectively, to simulate these other formats.
- You should probably disable character matching before executing an IOENTERB or IOENTERAB because the character in the "match" string is generally a valid binary value, rather than a termination character.
- The commands that are sent to the scope are device dependent and are found in the manual for the scope.
- The error checking in the program consists of executing an IF statement after each call to an HP-IB command.
- Before an IOENTERS statement is executed, space should be allocated for the string by assigning the string to SPACE$(*n*) where *n* is the maximum

length to be entered. If this is not done, the string will be of length 0, and no characters will be entered. You can then shorten the string to the correct length (in case less than the maximum number of characters were entered) by using the LEFT$ function.

The program has three main parts to it:

1. Read the data from the scope.
2. Print some statistics about the data.
3. Have the scope send the data to a printer.

```
'This program tells the scope to take a reading on channel 1, then
'sends the data back to this program.  We can do anything we want
'to the data at this time, and we choose to print some simple statistics
'about the data.  The program then tells the scope to send the data
'directly to the printer, illustrating how the controller doesn't have
'to be directly involved in an HP-IB transaction.
'
REM $INCLUDE: 'QBSETUP'
ISC& = 7
SCOPE& = ISC& * 100 + 7
'
'reset the HPIB interface
'
CALL IORESET(ISC&)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'set up a timeout of 5 seconds
'
TIME! = 5
CALL IOTIMEOUT(ISC&, TIME!)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'clear the scope
'
CALL IOCLEAR(SCOPE&)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'lockout the keyboard on the scope so the program has complete control
' of it
'
CALL IOREMOTE(ISC&)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'set up some variables
'
OPTION BASE 1
DISABLE% = 0 : ENABLE% = 1
EOL$ = CHR$(10)
CRLF$ = CHR$(13) + CHR$(10)
CMD$ = SPACE$(255)
'
```

```
'
'
'this is the beginning of part 1    ************************
'      - get the data from the scope
'
'
'setup scope to accept waveform data
'
CMD$ = "*RST"
LENGTH% = LEN(CMD$)
CALL IOOUTPUTS(SCOPE&, CMD$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
CMD$ = ":autoscale"
LENGTH% = LEN(CMD$)
CALL IOOUTPUTS(SCOPE&, CMD$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'setup up the waveform source
'
CMD$ = ":waveform:format word"
LENGTH% = LEN(CMD$)
CALL IOOUTPUTS(SCOPE&, CMD$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'input waveform preamble to controller
'
CMD$ = ":digitize channel1"
LENGTH% = LEN(CMD$)
CALL IOOUTPUTS(SCOPE&, CMD$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
CMD$ = ":waveform:preamble?"
LENGTH% = LEN(CMD$)
CALL IOOUTPUTS(SCOPE&, CMD$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'read in the preamble
'
MAX% = 10: ACTUAL% = 0
DIM PRE![10]
```

```
CALL IOENTERA(SCOPE&, SEG PRE![1], MAX%, ACTUAL%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'turn off 'lf' enter terminator, as 'lf' is a legal binary value
'
CALL IOMATCH(ISC&, EOL$, DISABLE%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'make sure EOI is set
'
CALL IOEOI(ISC&, ENABLE%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'command scope to send data.  It will be sent in IEEE arbitrary block
'format.
'
CMD$ = ":waveform:data?"
LENGTH% = LEN(CMD$)
CALL IOOUTPUTS(SCOPE&, CMD$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'enter the data
'
DIM READINGS%(5000): BYTES% = 8000: ACTUAL% = 0: FLAG% = 2
CALL IOENTERAB(SCOPE&, SEG READINGS%(1), BYTES%, ACTUAL%, FLAG%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'use IOGETTERM to see if we entered all the points
'
CALL IOGETTERM(ISC&, REASON%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
IF (REASON% AND 1) = 0 THEN PRINT "NOT ALL POINTS FOUND"
'
' Read the last byte from the scope.  This must always be done
' after an IOENTERAB command.  If the character read is a
' 'lf', then the device is done sending data.  If the character
' read is a ';' or a ',', then the device is waiting to send
' another block of data.
'
' Note also that we can use the select code instead of the device
' address for the first parameter of this command.  This is because
```

Computer Museum

```
' the scope is still addressed to talk, and the computer to listen
' from the IOENTERAB command.
'
LENGTH% = 1: ACTUAL% = 0
CMD$ = SPACE$(10)
CALL IOENTERS (ISC&, CMD$, LENGTH%, ACTUAL%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
CMD$ = LEFT$(CMD$, ACTUAL%)
'
IF CMD$ <> CHR$(10) THEN PRINT "scope wants to send more data..."
'
'
'this is the beginning of part 2    **************************
'     - print some statistics about the data
'
'
'calculate minimum, maximum, and sum of the values in the data
'
VDIV!   = 32 * PRE(8)
OFFSET! = (128 - PRE(10)) * PRE (8) + PRE (9)
SDIV!   = PRE(3) * PRE (5) / 10
DELAY!  = (PRE(3) / 2 - PRE(7)) * PRE(5) + PRE(6)
'
'Retrieve the scope's ID string
'
CMD$ = "*IDN?"
LENGTH% = LEN(CMD$)
CALL IOOUTPUTS(SCOPE&, CMD$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
LENGTH% = 50: ACTUAL% = 0
CMD$ = SPACE$(50)
CALL IOENTERS (SCOPE&, CMD$, LENGTH%, ACTUAL%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'print the stats
'
PRINT
PRINT " Oscilloscope ID: ";CMD$
PRINT " -------  statistics  --------"
PRINT "      Volts/Div = "; VDIV!;   " V"
PRINT "          Offset = "; OFFSET!; " V"
PRINT "           S/Div = "; SDIV!;   " S"
PRINT "           Delay = "; DELAY!;  " S"
PRINT
'
'
'
'this is the beginning of part 3    ************************
'     - have the scope send the graph directly to the printer
'
'
'Next, let's tell the scope to print directly to a printer.
' We must first send the HP-IB commands to make the scope a talker and
' the printer a listener.  This is done with the IOSEND command.
' We will tell the scope to issue a service request when it's done
' printing, as we need to wait for the printing to complete before
' continuing the program.
'
'tell the scope to SRQ on 'operation complete'
'
CMD$ = "*SRE 32 ; *ESE 1"
LENGTH% = LEN(CMD$)
CALL IOOUTPUTS(SCOPE&, CMD$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'tell scope to print
'
CMD$ = ":print? ; *OPC"
LENGTH% = LEN(CMD$)
CALL IOOUTPUTS(SCOPE&, CMD$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'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
'
PRINT.LISTEN$ = CHR$(32 + 1)   'printer is at device address 1
SCOPE.TALK$   = CHR$(64 + 7)   'scope is at device address 7
UNLISTEN$ = CHR$(63)
```

```
'
'send the command
'
CMD$ = UNLISTEN$ + PRINT.LISTEN$ + SCOPE.TALK$
LENGTH% = LEN(CMD$)
CALL IOSEND(ISC&, CMD$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'now, the ATN line must be set to FALSE.
'
COND% = 8
STATUS% = 0
CALL IOCONTROL(ISC&, COND%, STATUS%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'
'now wait for SRQ from scope
'
NOT.DONE:
'
COND% = 1: STATUS% = 0
CALL IOSTATUS(ISC&, COND%, STATUS%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
IF STATUS% = 0 THEN GOTO NOT.DONE
'
'make sure it was the scope requesting service
'
CALL IOSPOLL(SCOPE&, STATUS%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
' 64 = bit 6 set
'
IF (STATUS% AND 64) = 0 THEN GOTO NOT.DONE
'
' Clear the status byte so the scope can assert SRQ again
' if needed
'

CMD$ = "*CLS"
LENGTH% = LEN(CMD$)
```

```
CALL IOOUTPUTS(SCOPE&, CMD$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'give local control back to the scope
'
CALL IOLOCAL(ISC&)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
SYSTEM
```

Multimeter Example

The following program is written in QuickBASIC 4.5..

```
 ' This example uses the HP 34401A Multimeter as the primary device.
 ' We will also use the HP 3325A Function Generator as a source for
 ' the multimeter.
 '
 ' This example sets up the meter to take 128 readings, reads the data
 ' into an array, then plots the data on the screen.  In effect, it
 ' turns the multimeter into a simple oscilloscope.  This program is
 ' also checking other devices that are on the bus to see if they need
 ' service.  The SRQ line along with parallel and serial polling is
 ' used to make these checks.  The program will continue until the user
 ' presses the F1 key on the PC keyboard.
 '
 '
 REM $INCLUDE: 'QBSETUP'
 '
 NUM.READINGS = 128
 ISC& = 7                     'interface select code
 SOURCE& = ISC& * 100 + 12    'address of the function generator
 DVM& = ISC& * 100 + 22       'address of the digital volt meter
 DIM READINGS![128]           'place we will put the readings from the dvm
 '
 'the F1 key will end the program
 '
 ON KEY(1) GOSUB QUIT
 KEY(1) ON
 '
```

```
'reset the HPIB interface
'
CALL IORESET(ISC&)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'set up a timeout of 3 seconds
'
TIME = 3
CALL IOTIMEOUT(ISC&, TIME)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'clear the devices we are going to use
'
CALL IOCLEAR(SOURCE&)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
CALL IOCLEAR(DVM&)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'we meet the minimum requirements for IOFASTOUT, so let's use it
'
CALL IOFASTOUT(ISC&, TRUE%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
NULL$ = ""
CALL IOEOL(ISC&, NULL$, FALSE%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'we will now configure all devices that can respond to a parallel poll
' this example assumes devices at addresses 20 and 7 can respond to a
' parallel poll.  see operators manual of individual devices to see if
' they can respond to a parallel poll.
'
'configure the device at address 20 for a parallel poll
'
DEVICE.ADDR& = ISC& * 100 + 20
CONFIGURATION% = &H08 'RESPOND WITH A "1" ON LINE 0
CALL IOPPOLLC(DEVICE.ADDR&, CONFIGURATION%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'configure the device at address 7 for a parallel poll
```

```
'
DEVICE.ADDR& = ISC& * 100 + 7
CONFIGURATION% = &H09 'RESPOND WITH A "1" ON LINE 1
CALL IOPPOLLC(DEVICE.ADDR&, CONFIGURATION%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'configure any other devices that can respond to parallel poll here
'
'let's use dma to send the strings to program the devices
'
COUNT% = 40: CHANNEL% = 3
CALL IODMA(ISC&, COUNT%, CHANNEL%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'program the function generator
'
CODES$ = "RF1 FR30HZ FU1 ST1KH SP10KH MF1KH AM1VR TI5SE"
LENGTH% = LEN(CODES$)
CALL IOOUTPUTS(SOURCE&, CODES$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'program the dvm
'
CODES$ = ":CONF:VOLT:DC 30,.1;"
CODES$ = CODES$ + ":ZERO:AUTO OFF;"
CODES$ = CODES$ + ":TRIG:DELAY MIN;"
CODES$ = CODES$ + ":DISP:STATE OFF;"
LENGTH% = LEN(CODES$)
CALL IOOUTPUTS (DVM&, CODES$, LENGTH%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'turn dma off again
'
COUNT% = 0: CHANNEL% = 3
CALL IODMA(ISC&, COUNT%, CHANNEL%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'turn on automatic srq checking
'
ON PEN GOSUB SRQ.HANDLER
PEN ON
```

```
'
PRIORITY% = 0
CALL IOPEN(ISC&, PRIORITY%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
'
'
BACK:
    '
    'can have controller do other work here
    '
    GOSUB GET.DATA
    '
    GOTO BACK
'
'end of main program.  support routines follow.
'
'
'this is the routine defined by the 'on pen gosub' statement above.
' this routine will be called each time an srq comes in.
'
SRQ.HANDLER:
    '
    'conduct a parallel poll
    'note that the source doesn't respond to parallel poll's, so we
    ' need to poll that device separately.
    '
    RESPONSE% = 0
    CALL IOPPOLL(ISC&, RESPONSE%)
    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
    IF (RESPONSE% AND 1) THEN GOSUB POLL.DEVICE.1
    IF (RESPONSE% AND 2) THEN GOSUB POLL.DEVICE.2
    '
    'check all devices that were configured to respond to parallel
    ' poll
    '
    'check any other devices on the bus here that weren't configured
    ' to respond to parallel poll by performing a serial poll on each
    ' one.
    '
    CALL IOSPOLL(SOURCE&, RESPONSE%)
    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
    '
    'see if we've cleared the SRQ yet
    '
    STAT% = 1: RESPONSE% = 0
    CALL IOSTATUS(ISC&, STAT%, RESPONSE%)
    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
    IF RESPONSE% = 1 THEN PRINT "SRQ LOCKED HIGH" : GOTO QUIT
    '
RETURN
'
'
POLL.DEVICE.1:
    '
    'do a serial poll of the device configured to use parallel poll
    ' line 0
    '
    DEVICE.ADDR& = ISC& * 100 + 20
    CALL IOSPOLL(DEVICE.ADDR&, RESPONSE%)
    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
    '
    'should check RESPONSE% here to see if any action needs to be taken.
    'the values that RESPONSE% can take are device dependent.
    '
    RETURN
'
'
POLL.DEVICE.2:
    '
    'do a serial poll of the device configured to use parallel poll
    ' line 1
    '
    DEVICE.ADDR& = ISC& * 100 + 7
    CALL IOSPOLL(DEVICE.ADDR&, RESPONSE%)
    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
    '
    'should check RESPONSE% here to see if any action needs to be taken.
    'the values that RESPONSE% can take are device dependent.
    '
    RETURN
'
'
```

```
GET.DATA:
    '
    'Ask the DVM to send us the data
    '
    CODES$ = ":SAMPLE:COUNT 128;"
    CODES$ = CODES$ + ":READ?"
    LENGTH% = LEN(CODES$)
    CALL IOOUTPUTS (DVM&, CODES$, LENGTH%)
    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
    '
    'Read in the data
    '
    LENGTH% = NUM.READINGS
    ACTUAL% = 0
    CALL IOENTERA (DVM&, SEG READINGS! (0), LENGTH%, ACTUAL%)
    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
    '
    'set graphics mode and draw border on screen
    '
    CLS: SCREEN 2
    WINDOW (0,0)-(639,199)
    LINE (0,0)-(639,199),,B
    '
    'calculate min and max values for y-axis
    '
    YMAX = READINGS![0] : YMIN = READINGS![0]
    FOR I = 1 TO ACTUAL%
        IF READINGS![I] < YMIN THEN YMIN = READINGS![I]
        IF READINGS![I] > YMAX THEN YMAX = READINGS![I]
    NEXT I
    '
    'print graph labels
    '
    LOCATE 2,2 : PRINT "MAX = ";YMAX;
    LOCATE 24,2 : PRINT "MIN = ";YMIN;
    '
    'scale ymin and ymax so there is space between graph and border
    '
    IF YMIN > 0 THEN YMIN = YMIN*0.6 ELSE YMIN = YMIN*1.4
    IF YMAX > 0 THEN YMAX = YMAX*1.4 ELSE YMAX = YMAX*0.6
    '
    'graph the data
    '
    WINDOW (0,YMIN)-(ACTUAL%-1,YMAX)
    XAXIS = 1
    PREV = READINGS![XAXIS-1]
    PSET(XAXIS-1, PREV)
    WHILE (XAXIS < ACTUAL%)
      CURRENT = READINGS![XAXIS]
      LINE (XAXIS-1, PREV)-(XAXIS,CURRENT)
      PREV = CURRENT
      XAXIS = XAXIS + 1
    WEND
    RETURN
'
'
QUIT:
    '
    'clear the dvm so we can send the commands to reset it
    '
    CALL IOCLEAR(DVM&)
    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
    '
    'reset the dvm
    '
    CODES$ = ":DISP:STATE ON; *RST"
    LENGTH% = LEN(CODES$)
    CALL IOOUTPUTS (DVM&, CODES$, LENGTH%)
    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
    '
    'unconfigure the parallel poll
    '
    CALL IOPPOLLU(ISC&)
    IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
    '
    SYSTEM
```

4

BASIC Reference

This chapter presents a detailed Command Library syntax reference for GW-BASIC and QuickBASIC languages.

Parameters for Library commands are separated into several groups according to the types of arguments you must provide. The following table summarizes these groups. See "Command Library Parameters" in chapters 2 and 3 for more detail about parameter types for GW-BASIC and QuickBASIC.

Parameter Type	GW-BASIC	QuickBASIC and QBasic
Select Codes and Addresses	Single-precision real variable	Long-integer variable
Flags and Discrete Information	Single-precision real variable	Integer variable
Numeric Data (Single)	Single-precision real variable	Single-precision real variable
Numeric Data (Array)	Single-precision real array	Single-precision real array passed by far reference
Binary Data (Array)	Any type of numeric array	Any type of numeric array passed by far reference
String and Character Data	String variable	String variable

For QuickBASIC 4.0 and later, QBasic, and Microsoft Compiled BASIC, you can also use literals and expressions for simple parameters that provide information to the command—but *not* for parameters that return information.

Throughout this chapter, HP-IB terms are listed by abbreviation rather than by name. For example, “Go To Local” is listed as “GTL.” A complete list of HP-IB abbreviations is included in appendix B, “Summary of HP-IB.”

IOABORT

This command aborts all activity on the interface. IOABORT will abort as much as it can depending upon its current system controller and active controller status.

Syntax

`IOABORT (`*select_code*`)`

select_code	specifies the interface select code.

Examples

For GW-BASIC:

```
1100 DEV = 7
1200 CALL IOABORT(DEV)
1300 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

For QuickBASIC and QBasic:

```
DEV& = 7
CALL IOABORT(DEV&)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

Bus Activity

If the HP 82335 is system controller:

- IFC is pulsed at least 100 microseconds.
- REN is set.
- ATN is cleared.

If the HP 82335 is active, but not system controller:

- UNT is sent.

If the HP 82335 is neither active nor system controller:

- No bus activity.

Comments

Devices in Local Lockout will remain locked out.

Possible errors are NOERR and ESEL.

If the HP 82335 was the system, but not active controller, IOABORT will make the HP 82335 both system and active controller.

IOCLEAR

This command returns a device to a known, device-dependent state. It can be addressed to the interface or to a specific device.

Syntax

IOCLEAR (*device_address*)

IOCLEAR (*select_code*)

device_address	specifies the address of a device to be cleared.
select_code	specifies the select code of the interface on which all devices are to be cleared.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 DVM = 723
1120 CALL IOCLEAR(DVM) 'Clear the device at address 23.
.
.
1150 CALL IOCLEAR(ISC) 'Clear all devices on the interface.
```

For QuickBASIC and QBasic:

```
ISC& = 7
DVM& = 723
CALL IOCLEAR(DVM&) 'Clear the device at address 23.
  .
  .
CALL IOCLEAR(ISC&) 'Clear all devices on the interface.
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- SDC is sent.

If a select code is specified:

- ATN is set.
- DCL is sent.

Comments

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

IOCONTROL

This command directly sets status conditions in the interface. It can be used to address or unaddress the interface as a talker or listener, or set the interface's bus address. IOCONTROL can also change system controller status of the HP 82335 interface.

Note

IOCONTROL should be used with caution since it operates directly on the interface.

Syntax

`IOCONTROL` (*select_code*, *condition*, *status*)

select_code	specifies the interface select code.
condition	specifies the status condition that is to be set. Conditions which can be set are:

Value	**Description**
3	Make the interface the non-system or system controller.
5	Address or unaddress the interface as talker.
6	Address or unaddress the interface as listener.
7	Set the interface's bus address.
8	Clear or set ATN.

status variable into which the condition's status is placed. It can have the following values:

Condition 3

Value	Meaning
0	Make interface non-system controller
1	Make interface system controller

Conditions 5 and 6

Value	Meaning
0	Clear this condition
1	Set specified condition

Condition 7

Value	Meaning
0 to 30	Bus address of interface

Condition 8

Value	Meaning
0	Clear ATN
1	Set ATN asynchronously
2	Set ATN synchronously
Other	ERANGE error

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 COND = 5
1120 STATUS = 1
1130 CALL IOCONTROL(ISC,COND,STATUS)
1140 'Address the interface as talker
```

For QuickBASIC and QBasic:

```
ISC& = 7
COND% = 5
STATUS% = 1
CALL IOCONTROL(ISC&,COND%,STATUS%)
'Address the interface as talker
```

Bus Activity

None.

Comments

The added functionality for changing system controller status of the HP 82335 is included for completeness in the Command Library. HP strongly recommends, however, that you do not use this command unless it is absolutely necessary. The recommended method of using the interface as a non-system controller is to use the DOS command SYSCTL.EXE in your AUTOEXEC.BAT file as described in Chapter 1.

Refer to the Comments section of the IOPASSCTL command for important information about using Ctrl-C and Ctrl-Break.

For condition 8, you can set ATN either synchronously or asynchronously. Typically, you will set ATN asynchronously. If so, data may get lost if a data transfer is occurring that does not involve the HP 82335. For example, if a scope is talking to a printer and ATN is set asynchronously, some data may have been lost. If you want to avoid this situation, use status 2 to set ATN synchronously.

Possible errors are NOERR, ESEL, ECTRL, ETIME, and ERANGE.

IODMA

This command sets up DMA control. Using DMA may decrease the time required to transfer longer sequences of data using IOENTERAB, IOENTERB, IOENTERS, IOOUTPUTAB, IOOUTPUTB, and IOOUTPUTS.

Syntax

`IODMA (`*select_code*`,`*value*`,`*channel*`)`

select_code	specifies the interface on which DMA is to be enabled or disabled.
value	specifies one of the following:

Value	**Action Taken**
zero	Disables DMA. This is the default value.
positive value	Transfer size. Determines when a DMA read or write is executed. For example, if *value* = 100, then DMA will be used when 100 or more bytes are to be read or written.
negative value	Illegal. Will return an error.

channel	indicates which channel to use for DMA. If the channel is other than 2 or 3, an error is returned.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 VALUE = 1000
1120 CHANNEL = 3
1130 CALL IODMA(ISC,VALUE,CHANNEL)
```

For QuickBASIC and QBasic:

```
ISC& = 7
VALUE% = 1000
CHANNEL% = 3
CALL IODMA(ISC&,VALUE%,CHANNEL%)
```

Bus Activity

None.

Comments

DMA channel 3 is the recommended channel. This is least likely to conflict with established usage.

If character matching is enabled at the time IOENTERAB, IOENTERB, or IOENTERS using DMA is attempted, the error EUNKNOWN will be returned for that command and no data will be transferred.

If byte swapping is specified in IOENTERAB, IOENTERB, IOOUTPUTAB, or IOOUTPUTB using DMA (*swapsize* is greater than 1), the error EUNKNOWN will be returned for that command and no data will be transferred.

Possible errors are NOERR, ESEL, and ERANGE.

IOENTER

This command reads a single real number. Reading continues until one of these events occurs:

- The EOI line is sensed true, if it is enabled.
- A linefeed is encountered after the number starts.

Numeric characters are the digits 0 through 9, "E", "e", "+", "−", and "." in the proper sequence for representing a number. Note that " " (space) is not a numeric character.

Syntax

IOENTER (*device_address*, *data*)
IOENTER (*select_code*, *data*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data	variable into which the reading is placed.

Examples

For GW-BASIC:

```
1100 DEVICE = 722
1110 CALL IOENTER(DEVICE,READING)
1120 'Input a number from device 722 and place it in READING
```

For QuickBASIC and QBasic:

```
DEVICE& = 722
CALL IOENTER(DEVICE&,READING!)
'Input a number from device 722 and place it in READING
```

Bus Activity

If a device address is specified:

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.

If a select code is specified:

- If the interface is not addressed to listen, an error results.
- If the interface is addressed to listen, ATN is cleared and the data is read from the interface.

Comments

If a select code is to be specified in the command, the interface must first be addressed to listen (with IOSEND, for example) or an error occurs.

The approximate range of valid values is 10^{-38} to 10^{38}. The IEEE 754 standard for floating point numbers makes provisions for values less than 10^{-38}, however the internal number conversion may not properly handle values less than 10^{-38} when entered via HP-IB or used in assignment or print statements.

Possible errors are NOERR, ETIME, ESEL, EADDR, ECTRL, and ENUM.

IOENTERA

This command enters numbers from a device or the interface and places them into a real array. Reading continues until one of these events occurs:

- The EOI line is sensed true, if it is enabled.
- A linefeed is encountered after the specified number of elements is received.

Numeric characters are the digits 0 through 9, "E", "e", "+", "−", and "." in the proper sequence for representing a number. Note that " " (space) is not a numeric character.

Syntax

For GW-BASIC:

`IOENTERA` (*device_address*, *readings*, *max.elements*, *actual.elements*)
`IOENTERA` (*select_code*, *readings*, *max.elements*, *actual.elements*)

For QuickBASIC

`IOENTERA` (*device_address*,`SEG` *readings*, *max.elements*, *actual.elements*)
`IOENTERA` (*select_code*,`SEG` *readings*, *max.elements*, *actual.elements*)

device_address	specifies a device address.
select_code	specifies the interface select code.
readings	array into which the readings are placed.
max.elements	specifies the maximum number of elements to be read. (An error occurs if the number is less than 0.)
actual.elements	variable returning the number of elements actually read.

For QBasic:

`IOENTERA` (*device_address*, *segment*, *offset*, *max.elements*, *actual.elements*)
`IOENTERA` (*select_code*, *segment*, *offset*, *max.elements*, *actual.elements*)

device_address	specifies a device address.
select_code	specifies the interface select code.
segment	segment of the data array (use `VARSEG(`*array*`)`).
offset	offset of the data array (use `VARPTR(`*array*`)`).
max.elements	specifies the maximum number of elements to be read. (An error occurs if the number is less than 0.)
actual.elements	variable returning the number of elements actually read.

Examples

For GW-BASIC:

```
1100 DIM READINGS(49)
1110 DEVICE = 723
1120 MAX.ELEM = 50 : ACT.ELEM = 0
1130 CALL IOENTERA(DEVICE,READINGS(0),MAX.ELEM,ACT.ELEM)
1140 'Read a maximum of 50 values from device 723 and
1150 'put them in READINGS.
```

For QuickBASIC

```
DIM READINGS!(49)
DEVICE& = 723
MAX.ELEM% = 50 : ACT.ELEM% = 0
CALL IOENTERA(DEVICE&,SEG READINGS!(0),MAX.ELEM%,ACT.ELEM%)
'Read a maximum of 50 values from device 723 and
'put them in READINGS.
```

For QBasic:

```
DIM READINGS!(49)
DEVICE& = 723
MAX.ELEM% = 50 : ACT.ELEM% = 0
CALL IOENTERA(DEVICE&,VARSEG(READINGS!(0)),VARPTR(READINGS!(0),
                                           MAX.ELEM%,ACT.ELEM%)
'Read a maximum of 50 values from device 723 and
'put them in READINGS.
```

Bus Activity

If a device address is specified:

Computer Museum

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.

If a select code is specified:

- If the interface is not addressed to listen, an error results.
- If the interface is addressed to listen, ATN is cleared and the data is read from the interface.

Comments

You should initialize the *actual.elements* parameter to zero before you use it in an IOENTERA command.

If the specified maximum number of elements to read is greater than the size of the *readings* array, input data can overrun the array and corrupt existing data or programs.

Nonnumeric characters that do not properly belong in a real number are considered value separators. Thus, the sequence "1,234,567" is entered as three numbers, not as "1234567".

The number of readings available is dependent upon the source device.

The approximate range of valid values is 10^{-38} to 10^{38}. The IEEE 754 standard for floating point numbers makes provisions for values less than 10^{-38}, however the internal number conversion may not properly handle values less than 10^{-38} when entered via HP-IB or used in assignment or print statements.

If a select code is to be specified in the command, the interface must first be addressed to listen (with IOSEND, for example) or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ENUM, ECTRL, and ERANGE.

IOENTERAB

This command enters arbitrary-block program data (numeric data with IEEE-488.2 coding) from a device or the interface. Reading continues until one of these events occurs:

- The maximum number of bytes specified is received.
- A linefeed is encountered with the EOI line sensed true, if the coding indicates indefinite length.
- The number of bytes indicated by the coding is received, if the coding indicates definite length.

Syntax

For GW-BASIC:

`IOENTERAB` (*device_address*, *data*, *max.bytes*, *actual.bytes*, *swapsize*)
`IOENTERAB` (*select_code*, *data*, *max.bytes*, *actual.bytes*, *swapsize*)

For QuickBASIC:

`IOENTERAB` (*device_address*,`SEG` *data*, *max.bytes*, *actual.bytes*, *swapsize*)
`IOENTERAB` (*select_code*,`SEG` *data*, *max.bytes*, *actual.bytes*, *swapsize*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data	array into which the readings are placed.
max.bytes	specifies the maximum number of bytes to be read (excluding the coding bytes). (An error occurs if the number is less than 0.)
actual.bytes	variable returning the number of bytes actually read (excluding the coding bytes).
swapsize	specifies how bytes are placed into memory. A value of 1 indicates that bytes are placed in order. Larger values indicate that bytes are reversed in memory in groups of this size. The value should correspond to the byte size of the *data* variable. (For example, a value of 4 specifies that each

group of four bytes is swapped in memory.) Valid values are 1 through 8—other values return an error.

For QBasic:

IOENTERAB (*device_address*, *segment*, *offset*, *max.elements*, *actual.elements*, *swap*)
IOENTERAB (*select_code*, *segment*, *offset*, *max.elements*, *actual.elements*, *swap*)

device_address	specifies a device address.
select_code	specifies the interface select code.
segment	segment of the data array (use VARSEG(*array*)).
offset	offset of the data array (use VARPTR(*array*)).
max.elements	specifies the maximum number of elements to be read. (An error occurs if the number is less than 0.)
actual.elements	variable returning the number of elements actually read.
swap	specifies how bytes are placed into memory. A value of 1 indicates that bytes are placed in order. Larger values indicate that bytes are reversed in memory in groups of this size. The value should correspond to the byte size of the *data* variable. (For example, a value of 4 specifies that each group of four bytes is swapped in memory.) Valid values are 1 through 8—other values return an error.

Examples

For GW-BASIC:

```
1100 DIM READINGS#(49) 'Double-precision array (8 bytes/elem)
1110 DEVICE = 723
1120 SWAP = 8 : MAX.BYTE = 50 * SWAP : ACT.BYTE = 0
1130 CALL IOENTERAB(DEVICE,READINGS#(0),MAX.BYTE,ACT.BYTE,SWAP)
1140 'Read a maximum of 50 values from device 723 and
1150 'put them in READINGS.
```

For QuickBASIC:

```
DIM VAL#(49) 'Double-precision array (8 bytes/elem)
DEVICE& = 723
SWAP% = 8 : MAX.BYTE% = 50 * SWAP% : ACT.BYTE% = 0
CALL IOENTERAB(DEVICE&,SEG VAL#(0),MAX.BYTE%,ACT.BYTE%,SWAP%)
'Read a maximum of 50 values from device 723 and
'put them in VAL.
```

For QBasic:

```
DIM VAL#(49) 'Double-precision array (8 bytes/elem)
DEVICE& = 723
SWAP% = 8 : MAX.BYTE% = 50 * SWAP% : ACT.BYTE% = 0
CALL IOENTERAB(DEVICE&,VARSEG(VAL#(0)),VARPTR(VAL#(0)),MAX.BYTE%,
                                                 ACT.BYTE%,SWAP%)
'Read a maximum of 50 values from device 723 and
'put them in VAL.
```

Bus Activity

If a device address is specified:

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.

If a select code is specified:

- If the interface is not addressed to listen, an error results.
- If the interface is addressed to listen, ATN is cleared and the data is read from the interface.

Comments

IEEE-488.2 coding is described under "Arbitrary-Block Data Coding" in chapter 1. The coding bytes are not placed into *data*—this also applies to the ending linefeed character for indefinite-length data. Leading characters are ignored until a "#" character is received.

You should initialize the *actual.bytes* parameter to zero before you use it in an IOENTERAB command.

If the specified maximum number of elements to read is greater than the size of the *data* array, input data can overrun the array and corrupt existing data or programs.

If DMA is active for the transfer, the *swapsize* parameter must be 1 and character matching must be disabled—otherwise, an EUNKNOWN error occurs.

The number of bytes available is dependent upon the source device.

If a select code is to be specified in the command, the interface must first be addressed to listen (with IOSEND, for example) or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ERANGE, ECTRL, and EUNKNOWN.

IOENTERB

This command enters binary data (numeric data with no coding or formatting) from a device or the interface. Reading continues until one of these events occurs:

- The maximum number of bytes specified is received.
- The EOI line is sensed true, if it is enabled.
- The termination character set by IOMATCH is received with EOI true. (Linefeed is the default character.)

Syntax

For GW-BASIC:

`IOENTERB` (*device_address, data, max.bytes, actual.bytes, swapsize*)
`IOENTERB` (*select_code, data, max.bytes, actual.bytes, swapsize*)

For QuickBASIC

`IOENTERB` (*device_address*,`SEG` *data, max.bytes, actual.bytes, swapsize*)
`IOENTERB` (*select_code*,`SEG` *data, max.bytes, actual.bytes, swapsize*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data	array into which the readings are placed.
max.bytes	specifies the maximum number of bytes to be read. (An error occurs if the number is less than 0.)
actual.bytes	variable returning the number of bytes actually read.
swapsize	specifies how bytes are placed into memory. A value of 1 indicates that bytes are placed in order. Larger values indicate that bytes are reversed in memory in groups of this size. The value should correspond to the byte size of the *data* variable. (For example, a value of 4 specifies that each group of four bytes is swapped in memory.) Valid values are 1 through 8—other values return an error.

For QBasic:

`IOENTERB` (*device_address*, *segment*, *offset*, *max.elements*, *actual.elements*, *swap*)
`IOENTERB` (*select_code*, *segment*, *offset*, *max.elements*, *actual.elements*, *swap*)

device_address	specifies a device address.
select_code	specifies the interface select code.
segment	segment of the data array (use `VARSEG(`*array*`)`).
offset	offset of the data array (use `VARPTR(`*array*`)`).
max.elements	specifies the maximum number of elements to be read. (An error occurs if the number is less than 0.)
actual.elements	variable returning the number of elements actually read.
swap	specifies how bytes are placed into memory. A value of 1 indicates that bytes are placed in order. Larger values indicate that bytes are reversed in memory in groups of this size. The value should correspond to the byte size of the *data* variable. (For example, a value of 4 specifies that each group of four bytes is swapped in memory.) Valid values are 1 through 8—other values return an error.

Examples

For GW-BASIC:

```
1100 DIM READINGS#(49) 'Double-precision array (8 bytes/elem)
1110 DEVICE = 723
1120 SWAP = 8 : MAX.BYTE = 50 * SWAP : ACT.BYTE = 0
1130 CALL IOENTERB(DEVICE,READINGS#(0),MAX.BYTE,ACT.BYTE,SWAP)
1140 'Read a maximum of 50 values from device 723 and
1150 'put them in READINGS.
```

For QuickBASIC:

```
DIM VAL#(49) 'Double-precision array (8 bytes/elem)
DEVICE& = 723
SWAP% = 8 : MAX.BYTE% = 50 * SWAP% : ACT.BYTE% = 0
CALL IOENTERB(DEVICE&,SEG VAL#(0),MAX.BYTE%,ACT.BYTE%,SWAP%)
'Read a maximum of 50 values from device 723 and
'put them in VAL.
```

For QBasic:

```
DIM VAL#(49) 'Double-precision array (8 bytes/elem)
DEVICE& = 723
SWAP% = 8 : MAX.BYTE% = 50 * SWAP% : ACT.BYTE% = 0
CALL IOENTERB(DEVICE&,VARSEG(VAL#(0)),VARPTR(VAL#(0)),MAX.BYTE%,
                                                ACT.BYTE%,SWAP%)
'Read a maximum of 50 values from device 723 and
'put them in VAL.
```

Bus Activity

If a device address is specified:

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.

If a select code is specified:

- If the interface is not addressed to listen, an error results.
- If the interface is addressed to listen, ATN is cleared and the data is read from the interface.

Comments

You should initialize the *actual.bytes* parameter to zero before you use it in an IOENTERB command.

If the specified maximum number of elements to read is greater than the size of the *data* array, input data can overrun the array and corrupt existing data or programs.

All data received is stored in memory—except a final "match" character with EOI true if matching is enabled.

If DMA is active for the transfer, the *swapsize* parameter must be 1 and character matching must be disabled—otherwise, an EUNKNOWN error occurs.

The number of bytes available is dependent upon the source device.

If a select code is to be specified in the command, the interface must first be addressed to listen (with IOSEND, for example) or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ERANGE, ECTRL, and EUNKNOWN.

IOENTERF

This command reads from a device and places all received data into a file. Reading continues until one of these events occurs:

- The EOI line is sensed true, if it is enabled.
- The termination character set by IOMATCH is received (linefeed is the default). Note: If you are transferring binary files, you should turn off character match using IOMATCH to make sure the transfer does not end prematurely.
- The maximum number of bytes specified is received.
- A file error occurs, usually meaning the disk is full.

Syntax

`IOENTERF` (*device_address*, *file_name*, *length*, *append_flag*)
`IOENTERF` (*select_code*, *file_name*, *length*, *append_flag*)

device_address	specifies a device address.
select_code	specifies the interface select code.
file_name	specifies the file into which the data is written.
length	specifies the maximum number of elements to be read. (An error occurs if the number is less than 0.) The actual number of bytes read is returned here.
append_flag	specifies whether to append to the file or to overwrite it. Zero overwrites; non-zero appends.

Examples

For GW-BASIC:

```
1100 DEV = 723
1200 LENGTH = 10
1300 FILE.NAME$ = "ENTER.DAT"
1400 APPEND = 0
1500 CALL IOENTERF(DEV,FILE.NAME$,LENGTH,APPEND)
1600 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

For QuickBASIC and QBasic:

```
DEV& = 723
LENGTH& = 10
FILE.NAME$ = "ENTER.DAT"
APPEND% = 0
CALL IOENTERF(DEV&,FILE.NAME$,LENGTH&,APPEND%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

Bus Activity

If a device address is specified:

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.

If a select code is specified:

- If the interface is not addressed to listen, an error results.
- If the interface is addressed to listen, ATN is cleared and the data is read from the interface.

Comments

Possible errors are NOERR, ETIME, ESEL, EADDR, ERANGE, ECTRL, and EFILE.

Ifthe file does not exist, and a valid filename is given, IOENTERF will create the file regardless of the append flag.

We recommend turning off character matching using the IOMATCH command, especially if you are transferring a binary file.

Note

This command does **not** transfer files to an HP-IB disk drive, but rather transfers bytes from the HP-IB bus to a built-in disk drive on your computer.

IOENTERS

This command enters a character string from a device or the interface. Reading continues until one of these events occurs:

- The EOI line is sensed true, if it is enabled.
- The termination character set by IOMATCH is received (linefeed is the default).
- The maximum number of characters specified is received.

Syntax

IOENTERS (*device_address*, *data*, *max.length*, *actual.length*)
IOENTERS (*select_code*, *data*, *max.length*, *actual.length*)

device_address	specifies a device address.
selcct_code	specifies the interface select code.
max.length	specifies the maximum number of elements to be read. (An error occurs if the number is less than 0.)
actual.length	variable returning the number of elements actually read.

Examples

For GW-BASIC:

```
1100 DEV = 723
1110 MAX.LENGTH = 10 : ACTUAL.LENGTH = 0
1120 INFO$ = SPACE$(MAX.LENGTH)
1130 CALL IOENTERS(DEV,INFO$,MAX.LENGTH,ACTUAL.LENGTH)
1140 'Read a string of 10 characters maximum from
1150 'device 723, put in INFO$
1160 INFO$ = LEFT$(INFO$,ACTUAL.LENGTH)
```

For QuickBASIC and QBasic:

```
DEV& = 723
MAX.LENGTH% = 10 : ACTUAL.LENGTH% = 0
INFO$ = SPACE$(MAX.LENGTH%)
CALL IOENTERS(DEV&,INFO$,MAX.LENGTH%,ACTUAL.LENGTH%)
'Read a string of 10 characters maximum from
'device 723, put in INFO$
INFO$ = LEFT$(INFO$,ACTUAL.LENGTH%)
```

Bus Activity

If a device address is specified:

Computer Museum

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.

If a select code is specified:

- If the interface is not addressed to listen, an error results.
- If the interface is addressed to listen, ATN is cleared and the data is read from the interface.

Comments

You should initialize the *actual.bytes* parameter to zero before you use it in an IOENTERS command.

You should initialize the string into which data is read with the SPACE$ function before you call IOENTERS:

```
1150 INFO$ = SPACE$(10)    for GW-BASIC

INFO$ = SPACE$(10)         for QuickBASIC
```

This prevents early termination if the string was not initialized, or was set to some other value. If the specified maximum number of elements to read is greater than the current length of the *data* string, the current length is used instead of the maximum number.

To remove the termination character from the string, use the BASIC LEFT$ function. For example, to remove CR/LF from the end of your string, you can use this statement:

```
1260 INFO$ = LEFT$(INFO$,ACT.LENGTH-2)     for GW-BASIC

INFO$ = LEFT$(INFO$,ACT.LENGTH%-2)         for QuickBASIC
```

If a select code is to be specified in the command, the interface must first be addressed to listen (with IOSEND or a previous IOENTER, for example) or an error occurs.

The termination character is entered as part of the string.

If DMA is active for the transfer, character matching must be disabled—otherwise, an EUNKNOWN error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ENUM, ERANGE, ECTRL, and EUNKNOWN.

IOEOI

This command enables or disables the End Or Identify (EOI) mode of the interface. It is used to:

- Enable or disable a write operation to set the EOI line on the last byte of the write.
- Enable or disable a read operation to terminate upon sensing the EOI line true.

The default is EOI enabled.

Syntax

`IOEOI` (*select_code*, *state*)

select_code	specifies the interface select code.
state	enables EOI if nonzero and disables EOI if zero.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 STATE = 0
1120 CALL IOEOI(ISC,STATE) 'Disable EOI
```

For QuickBASIC and QBasic:

```
ISC& = 7
STATE% = 0
CALL IOEOI(ISC&,STATE%) 'Disable EOI
```

Bus Activity

None.

Comments

When reading with EOI enabled, receipt of a byte with EOI set causes the read operation to terminate, regardless of whether you are reading a string, a real number, or an array of real numbers. (The EOI state is ignored by IOENTERAB.)

When writing, EOI is set on the last byte of the End Of Line sequence if EOI is enabled. Note that if the EOL sequence is of 0 length, EOI is set on the last data byte sent. (The EOI line is *not* set on the last byte for IOOUTPUTAB.)

When sending a real number array with IOOUTPUTA, the EOL sequence (and subsequent EOI) is appended after the last element in the array, not after each element.

Note that IOSEND does not set EOI because this line has a different meaning in Command mode.

Possible errors are NOERR and ESEL.

IOEOL

This command defines the End of Line (EOL) string that is to be sent following every IOOUTPUT, IOOUTPUTA, IOOUTPUTB, and IOOUTPUTS command.

The default is carriage return and linefeed.

Syntax

`IOEOL` (*select_code*, *endline*, *length*)

select_code	specifies the interface select code.
endline	specifies the EOL string that is to be sent following a data transmission. A maximum of eight characters can be specified.
length	specifies the length of the termination string. If zero is specified, no characters are appended to a data transmission. If the length is less than 0 or more than 8, an error occurs.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 ENDLINE$ = CHR$(13)+CHR$(10)
1120 LENGTH = LEN(ENDLINE$)
1130 CALL IOEOL(ISC,ENDLINE$,LENGTH) 'EOL = CR/LF
```

For QuickBASIC and QBasic:

```
ISC& = 7
ENDLINE$ = CHR$(13)+CHR$(10)
LENGTH% = LEN(ENDLINE$)
CALL IOEOL(ISC&,ENDLINE$,LENGTH%) 'EOL = CR/LF
```

Bus Activity

None.

Comments

With IOOUTPUTA and IOOUTPUTB, the EOL sequence is appended after all data has been sent, not following each element.

Possible errors are NOERR, ESEL, and ERANGE.

IOFASTOUT

This command enables or disables high-speed bus timing for output transfers only.

The default is high-speed output disabled (standard speed).

Syntax

`IOFASTOUT` (*select_code*, *state*)

select_code	specifies the interface select code.
state	enables high-speed output if nonzero and disables high-speed output if zero.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 STATE = 0
1120 CALL IOFASTOUT(ISC,STATE) 'Disable high-speed output
```

For QuickBASIC and QBasic:

```
ISC& = 7
STATE% = 0
CALL IOFASTOUT(ISC&,STATE%) 'Disable high-speed output
```

Bus Activity

None.

Comments

For proper operation, high-speed output requires the HP-IB system to meet all of these requirements:

- All HP-IB devices must have tri-state drivers, not open-collector drivers. (The HP-IB interface meets this requirement.)
- All HP-IB devices must be turned on.
- HP-IB cable length should be as short as possible, but not longer than 15 meters (50 feet). At least one HP-IB device should be connected for each meter (3 feet) of cable, with a maximum of 15 devices. (The HP-IB interface counts as one device.)
- Each HP-IB device must have a capacitance of less than 50 pF on each HP-IB line except REN and IFC. (The HP-IB interface meets this requirement.)

High-speed output applies only during output transfers (including DMA output transfers)—but not between transfers and not during input transfers. The speed of an input transfer depends upon the talker device.

High-speed output decreases the data-settling time from 2.5 microseconds to 840 nanoseconds.

Possible errors are NOERR and ESEL.

IOGETTERM

This command determines the reason the last read terminated.

Syntax

`IOGETTERM` (*select_code*, *reason*)

select_code	specifies the interface select code.
reason	variable to receive the sum of the values for the reasons the last read terminated. The possible reasons for termination are

Value	**Description**
0	The read was terminated for some reason not covered by any of the other reasons.
1	The expected number of elements was received.
2	The termination character set by IOMATCH was encountered.
4	The EOI line was sensed true.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 CALL IOGETTERM(ISC,REASON)
1120 IF ((REASON AND 4) = 4) THEN PRINT "EOI ENCOUNTERED"
```

For QuickBASIC and QBasic:

```
ISC& = 7
CALL IOGETTERM(ISC&,REASON%)
IF ((REASON% AND 4) = 4) THEN PRINT "EOI ENCOUNTERED"
```

Bus Activity

None.

Comments

Upon return, the reason integer contains the sum of the values for the reasons for termination. For example, if the last read terminated when the termination character was encountered and EOI was set, the value of reason would be $2 + 4 = 6$.

Possible errors are NOERR and ESEL.

IOLLOCKOUT

This command sends a Local Lockout (LLO) to disable a device front panel. It is received by all devices on the interface, whether or not they are addressed to listen.

Syntax

IOLLOCKOUT (*select_code*)

select_code specifies the interface select code.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 CALL IOLLOCKOUT(ISC)
```

For QuickBASIC and QBasic:

```
ISC& = 7
CALL IOLLOCKOUT(ISC&)
```

Bus Activity

- ATN is sent.
- LLO is sent.

Comments

If a device is in Local mode when LLO is received, LLO does not take effect until the device is addressed to listen.

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

IOLOCAL

This command executes a Go To Local (GTL) or clears the REN line to enable a device front panel.

Syntax

`IOLOCAL` (*device_address*)
`IOLOCAL` (*select_code*)

device_address	specifies a device address.
select_code	specifies the interface select code.

Examples

For GW-BASIC:

```
1100 DEVICE = 722
1110 CALL IOLOCAL(DEVICE) 'Place device 722 in local mode.
```

For QuickBASIC and QBasic:

```
DEVICE& = 722
CALL IOLOCAL(DEVICE&) 'Place device 722 in local mode.
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- GTL is sent.

If a select code is specified:

- REN is cleared.
- ATN is cleared.

Comments

If a device address is specified, the device is temporarily placed in Local mode—it will return to Remote mode if it is later addressed to listen. If Local Lockout is in effect, the device will return to the Lockout state if it is later addressed to listen.

If an interface select code is specified, all instruments on the bus are placed in Local mode and any Local Lockout is cancelled.

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

IOMATCH

This command defines the character used by IOENTERB and IOENTERS for termination.

The default character is linefeed.

Syntax

`IOMATCH (`*select_code*`,`*character*`,`*flag*`)`

select_code	specifies the interface select code.
character	specifies the character used by IOENTERB and IOENTERS for termination checking.
flag	indicates whether character matching should be enabled or disabled. Zero disables matching, and any nonzero value enables it.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 MATCH$ = CHR$(10) 'Terminate on a linefeed.
1120 FLAG = 1
1130 CALL IOMATCH(ISC,MATCH$,FLAG)
```

For QuickBASIC and QBasic:

```
ISC& = 7
MATCH$ = CHR$(10) 'Terminate on a linefeed.
FLAG% = 1
CALL IOMATCH(ISC&,MATCH$,FLAG%)
```

Bus Activity

None.

Comments

Only a single match character may be specified in this command.

For IOENTERS, the match character becomes part of the entered string. For IOENTERB, the match character must be received with EOI true, and the character does *not* become part of the data.

IOMATCH does not apply to IOENTER, IOENTERA, or IOENTERAB.

Possible errors are NOERR and ESEL.

IOOUTPUT

This command outputs a real number to a device or to the interface. After the number is sent, the EOL string is sent and the EOI line is set (if enabled).

Syntax

IOOUTPUT (*device_address*, *data*)
IOOUTPUT (*select_code*, *data*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data	specifies the number to be output.

Examples

For GW-BASIC:

```
1100 INFO = 12.3
1110 DEV = 722
1120 CALL IOOUTPUT(DEV,INFO) 'output " 12.3" to device 722.
```

For QuickBASIC and QBasic:

```
INFO! = 12.3
DEV& = 722
CALL IOOUTPUT(DEV&,INFO!) 'output " 12.3" to device 722
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is output.
- EOL is output.

If a select code is specified:

- If the interface is not addressed to talk, an error results.
- If the interface is addressed to talk, ATN is cleared and the data is sent followed by an EOL.

Comments

Numbers with absolute values between 10^{-5} and 10^{6} are rounded to seven significant digits and output in floating point notation. If the number rounds to an integer value, the decimal point is not sent. Numbers outside this range are rounded to seven significant digits and output in scientific notation.

If the number is positive, a leading space is output for the sign; if it's negative, a leading "–" is output.

If a select code is to be specified, the interface must first be addressed to talk (with IOSEND, for example), or an error occurs.

Possible errors are NOERR, ETIME, ESEL, ECTRL, and EADDR.

IOOUTPUTA

This command outputs an array of real numbers to a specified device or to the bus. Values output are separated by commas. After the last number is sent, the EOL string is sent and the EOI line is set (if enabled).

Syntax

For GW-BASIC:

`IOOUTPUTA` (*device_address*, *data*, *elements*)
`IOOUTPUTA` (*select_code*, *data*, *elements*)

For QuickBASIC:

`IOOUTPUTA` (*device_address*,`SEG` *data*, *elements*)
`IOOUTPUTA` (*select_code*,`SEG` *data*, *elements*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data	array containing the real numbers to be transmitted.
elements	specifies the number of elements in the array to be transmitted. (An error occurs if the number is less than 0.)

For QBasic:

`IOOUTPUTA` (*device_address*, *segment*, *offset*, *elements*)
`IOOUTPUTA` (*select_code*, *segment*, *offset*, *elements*)

device_address	specifies a device address.
select_code	specifies the interface select code.
segment	segment of the data array (use `VARSEG(`*array*`)`).
offset	offset of the data array (use `VARPTR(`*array*`)`).
elements	specifies the number of elements in the array to be transmitted. (An error occurs if the number is less than 0.)

Examples

For GW-BASIC:

```
1100 DIM INFO(9)
1110 ELEMENTS = 10
1120 DEV = 722
1130 CALL IOOUTPUTA(DEV,INFO(0),ELEMENTS)
1400 'Output array INFO to device 722; begin with element 0.
```

For QuickBASIC:

```
DIM INFO!(9)
ELEMENTS% = 10
DEV& = 722
CALL IOOUTPUTA(DEV&,SEG INFO!(0),ELEMENTS%)
'Output array INFO to device 722; begin with element 0.
```

For QBasic:

```
DIM INFO!(9)
ELEMENTS% = 10
DEV& = 722
CALL IOOUTPUTA(DEV&,VARSEG(INFO!(0)),VARPTR(INFO!(0)),ELEMENTS%)
'Output array INFO to device 722; begin with element 0.
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is output.
- EOL is output.

If a select code is specified:

- If the interface is not addressed to talk, an error results.
- If the interface is addressed to talk, ATN is cleared and the data is sent followed by an EOL.

Comments

If the specified maximum number of elements to output is greater than the size of the *data* array, the output transfer can go beyond the array and send meaningless data.

Numbers with absolute values between 10^{-5} and 10^{6} are rounded to seven significant digits and output in floating point notation. If the number rounds to an integer value, the decimal point is not sent. Numbers outside this range are rounded to seven significant digits and output in scientific notation.

If the number is positive, a leading space is output for the sign; if it's negative, a leading "–" is output.

If a select code is to be used as a parameter, the interface must first be addressed to talk (with IOSEND, for example), or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ECTRL, and ERANGE.

IOOUTPUTAB

This command outputs arbitrary-block response data (numeric data with IEEE-488.2 coding) to a specified device or to the bus. After the last data byte is sent, nothing additional occurs.

Syntax

Computer Museum

For GW-BASIC:

IOOUTPUTAB (*device_address*, *data*, *bytes*, *swapsize*)
IOOUTPUTAB (*select_code*, *data*, *bytes*, *swapsize*)

For QuickBASIC:

IOOUTPUTAB (*device_address*,SEG *data*, *bytes*, *swapsize*)
IOOUTPUTAB (*select_code*,SEG *data*, *bytes*, *swapsize*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data	array containing the data to be transmitted.
bytes	specifies the number of bytes to output (excluding the coding bytes). This value should be no more than the number of elements in the array times the number of bytes per element. (An error occurs if the number is less than 0.)
swapsize	specifies how bytes are read from memory. A value of 1 indicates that bytes are read in order. Larger values indicate that bytes are reversed as read from memory in groups of this size. The value should correspond to the byte size of the *data* variable. (For example, a value of 4 specifies that each group of four bytes is swapped when output.) Valid values are 1 through 8—other values return an error.

For QBasic:

IOOUTPUTAB (*device_address*, *segment*, *offset*, *elements*, *swapsize*)
IOOUTPUTAB (*select_code*, *segment*, *offset*, *elements*, *swapsize*)

device_address	specifies a device address.

select_code	specifies the interface select code.
segment	segment of the data array (use `VARSEG(`*array*`)`).
offset	offset of the data array (use `VARPTR(`*array*`)`).
elements	specifies the number of elements in the array to be transmitted. (An error occurs if the number is less than 0.)
swapsize	specifies how bytes are read from memory. A value of 1 indicates that bytes are read in order. Larger values indicate that bytes are reversed as read from memory in groups of this size. The value should correspond to the byte size of the *data* variable. (For example, a value of 4 specifies that each group of four bytes is swapped when output.) Valid values are 1 through 8—other values return an error.

Examples

For GW-BASIC:

```
1100 DIM INFO#(9) 'Double-precision array (8 bytes/elem)
1110 SWAP = 8
1120 ELEMENTS = 10 * SWAP
1130 DEV = 722
1140 CALL IOOUTPUTAB(DEV,INFO#(0),ELEMENTS,SWAP)
1150 'Output array INFO to device 722; begin with element 0.
```

For QuickBASIC:

```
DIM INFO#(9) 'Double-precision array (8 bytes/elem)
SWAP% = 8
ELEMENTS% = 10 * SWAP%
DEV& = 722
CALL IOOUTPUTAB(DEV&,SEG INFO#(0),ELEMENTS%,SWAP%)
'Output array INFO to device 722; begin with element 0.
```

For QBasic:

```
DIM INFO#(9) 'Double-precision array (8 bytes/elem)
SWAP% = 8
ELEMENTS% = 10 * SWAP%
DEV& = 722
CALL IOOUTPUTAB(DEV&,VARSEG(INFO#(0)),VARPTR(INFO#(0)),ELEMENTS%,SWAP%)
'Output array INFO to device 722; begin with element 0.
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is output.

If a select code is specified:

- If the interface is not addressed to talk, an error results.
- If the interface is addressed to talk, ATN is cleared and the data is sent.

Comments

IEEE-488.2 coding is described under "Arbitrary-Block Data Coding" in chapter 1. The coding bytes are automatically computed and inserted in front of the data.

If the specified maximum number of elements to output is greater than the size of the *data* array, the output transfer can go beyond the array and send meaningless data.

If DMA is active for the transfer, the *swapsize* parameter must be 1—otherwise, an EUNKNOWN error occurs.

If a select code is to be specified in the command, the interface must first be addressed to talk (with IOSEND, for example) or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ERANGE, ECTRL, and EUNKNOWN.

IOOUTPUTB

This command outputs binary data (numeric data with no coding or formatting) to a specified device or to the bus. After the last number is sent, the EOL string is sent and the EOI line is set (if enabled).

Syntax

For GW-BASIC:

`IOOUTPUTB` (*device_address*, *data*, *bytes*, *swapsize*)
`IOOUTPUTB` (*select_code*, *data*, *bytes*, *swapsize*)

For QuickBASIC:

`IOOUTPUTB` (*device_address*,`SEG` *data*, *bytes*, *swapsize*)
`IOOUTPUTB` (*select_code*,`SEG` *data*, *bytes*, *swapsize*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data	array containing the data to be transmitted.
bytes	specifies the number of bytes to output. This value should be no more than the number of elements in the array times the number of bytes per element. (An error occurs if the number is less than 0.)
swapsize	specifies how bytes are read from memory. A value of 1 indicates that bytes are read in order. Larger values indicate that bytes are reversed as read from memory in groups of this size. The value should correspond to the byte size of the *data* variable. (For example, a value of 4 specifies that each group of four bytes is swapped when output.) Valid values are 1 through 8—other values return an error.

For QBasic:

`IOOUTPUTB` (*device_address*, *segment*, *offset*, *bytes*, *swapsize*)
`IOOUTPUTB` (*select_code*, *segment*, *offset*, *bytes*, *swapsize*)

device_address	specifies a device address.

select_code	specifies the interface select code.
segment	segment of the data array (use `VARSEG(`*array*`)`).
offset	offset of the data array (use `VARPTR(`*array*`)`).
bytes	specifies the number of bytes to output. This value should be no more than the number of elements in the array times the number of bytes per element. (An error occurs if the number is less than 0.)
swapsize	specifies how bytes are read from memory. A value of 1 indicates that bytes are read in order. Larger values indicate that bytes are reversed as read from memory in groups of this size. The value should correspond to the byte size of the *data* variable. (For example, a value of 4 specifies that each group of four bytes is swapped when output.) Valid values are 1 through 8—other values return an error.

Examples

For GW-BASIC:

```
1100 DIM INFO#(9) 'Double-precision array (8 bytes/elem)
1110 SWAP = 8
1120 ELEMENTS = 10 * SWAP
1130 DEV = 722
1140 CALL IOOUTPUTB(DEV,INFO#(0),ELEMENTS,SWAP)
1150 'Output array INFO to device 722; begin with element 0.
```

For QuickBASIC:

```
DIM INFO#(9) 'Double-precision array (8 bytes/elem)
SWAP% = 8
ELEMENTS% = 10 * SWAP%
DEV& = 722
CALL IOOUTPUTB(DEV&,SEG INFO#(0),ELEMENTS%,SWAP%)
'Output array INFO to device 722; begin with element 0.
```

For QBasic:

```
DIM INFO#(9) 'Double-precision array (8 bytes/elem)
SWAP% = 8
ELEMENTS% = 10 * SWAP%
DEV& = 722
CALL IOOUTPUTB(DEV&,VARSEG(INFO#(0)),VARPTR(INFO#(0)),ELEMENTS%,SWAP%)
'Output array INFO to device 722; begin with element 0.
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is output.
- EOL is output.

If a select code is specified:

- If the interface is not addressed to talk, an error results.
- If the interface is addressed to talk, ATN is cleared and the data is sent followed by an EOL.

Comments

If the specified maximum number of elements to output is greater than the size of the *data* array, the output transfer can go beyond the array and send meaningless data.

If DMA is active for the transfer, the *swapsize* parameter must be 1—otherwise, an EUNKNOWN error occurs.

If a select code is to be specified in the command, the interface must first be addressed to talk (with IOSEND, for example) or an EUNKNOWN error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ERANGE, ECTRL, and EUNKNOWN.

IOOUTPUTF

This command outputs the contents of a file to a specified device or interface. After the file is sent, the EOL string is sent and the EOI line is set (if enabled).

Syntax

IOOUTPUTF (*device_address*,*file_name*,*length*)
IOOUTPUTF (*select_code*,*file_name*,*length*)

device_address	specifies a device address.
select_code	specifies the interface select code.
file_name	specifies the name of the file to output.
length	specifies the maximum number of elements to be written. (An error occurs if the number is less than 0.)

Examples

For GW-BASIC:

```
1100 DEV = 723
1200 LENGTH = 10
1300 FILE.NAME$ = "OUTPUT.DAT"
1400 CALL IOOUTPUTF(DEV,FILE.NAME$,LENGTH)
1500 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

For QuickBASIC and QBasic:

```
DEV& = 723
LENGTH& = 10
CALL IOOUTPUTF(DEV&,FILE.NAME$,LENGTH&)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.
- EOL is output.

If a select code is specified:

- If the interface is not addressed to talk, an error results.
- If the interface is addressed to talk, ATN is cleared and the data is sent followed by the EOL string.

Comments

Possible errors are NOERR, ETIME, ESEL, EADDR, ERANGE, ECTRL, and EFILE.

If you are transferring a binary file, we recommend that you turn off the EOL string using the IOEOL command. If you do not, the current EOL string will be appended to the file.

Note

This command does **not** transfer files from an HP-IB disk drive, but rather transfers bytes from a built-in disk drive on your computer to the HP-IB bus.

IOOUTPUTS

This command outputs a string to a specified device or to the interface. After the string is sent, the EOL string is sent and the EOI line is set (if enabled).

Syntax

IOOUTPUTS (*device_address*, *data*, *length*)
IOOUTPUTS (*select_code*, *data*, *length*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data	array specifying the string to be sent.
length	specifies the length of the output string. (An error occurs if the number is less than 0.)

Examples

For GW-BASIC:

```
1100 DEV = 723
1110 INFO$ = "1ST1"
1120 LENGTH = LEN(INFO$)
1130 CALL IOOUTPUTS(DEV,INFO$,LENGTH)
1140 'Send "1ST1" to device 723.
```

For QuickBASIC and QBasic:

```
DEV& = 723
INFO$ = "1ST1"
LENGTH% = LEN(INFO$)
CALL IOOUTPUTS(DEV&,INFO$,LENGTH%)
'Send "1ST1" to device 723.
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is output.
- EOL is output.

If a select code is specified:

- If the interface is not addressed to talk, an error results.
- If the interface is addressed to talk, ATN is cleared and the data is sent followed by an EOL.

Comments

If the specified maximum number of elements to output is greater than the current length of the *data* string, the current length is used instead of the maximum number.

If a select code is to be used in the command, the interface must first be addressed to talk (with IOSEND, for example), or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ECTRL, and ERANGE.

IOPASSCTL

This command passes active control from the HP 82335 HP-IB card to a device on the bus. The device must be capable of taking control.

Syntax

`IOPASSCTL` (*device_address*)

device_address specifies a device address.

Examples

For GW-BASIC:

```
1100 KEY 15, CHR$(&H04) + CHR$(&H2E)    'Trap CTRL-C
1200 ON KEY(15) GOSUB 9000
1300 KEY(15) ON
1400 '
1500 DEV = 723
1600 CALL IOPASSCTL(DEV)
1700 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
...
9000 SYSTEM
```

For QuickBASIC:

```
KEY 15, CHR$(&H04) + CHR$(&H2E)   'Trap CTRL-C
ON KEY(15) GOSUB THEEND
KEY(15) ON

DEV& = 723
CALL IOPASSCTL(DEV&)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR

THEEND:  SYSTEM
```

For QBasic:

This command is not supported with QBasic.

Bus Activity

If a device address is specified:

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- TCT is sent.
- ATN is cleared.

Comments

If your program does not seem to work properly after passing control, make sure that you do not have an interrupt (IRQ) conflict with another device. You can find out what IRQ your HP-IB board is using by running the INSTALL utility.

The Command Library defaults to address 30. If you need to, you can change this using the IOCONTROL command.

The IOPASSCTL command passes active control only. This command will not change the state of the system controller status of the HP 82335.

Any type of *shell* command will cause the Command Library to stop working if it is currently non-controller, including the SHELL statement in BASIC languages.

The Command Library needs to do some cleaning up after running as non-active or non-system controller. It will do this automatically when you take control back, or when your program exits normally. It will not, however, clean up after itself if Ctrl-C or Ctrl-Break is used to exit the program. We recommend that you use the capabilities of the language you are using to trap these keys and call a routine which exits normally, possibly with a non-zero exit code. In BASIC, use *on key* and *system* if Ctrl-C is pressed. If you do not, your computer will be left in an unstable state and could lock up.

Possible errors are NOERR, ETIME, ESEL, and ECTRL.

IOPEN

This command enables the HP-IB interface to detect HP-IB service requests and process them as ON PEN events. It also sets the DMA-interrupt priority.

The default is service-request events disabled.

Syntax

`IOPEN` (*select_code*,*priority*)

select_code	specifies the interface select code.
priority	specifies how service requests affect DMA transfers. Valid values are listed below—other values are illegal and will result in an error.

Value	Description
0	Low priority—a service request won't interrupt a DMA transfer. This is the recommended value for general use.
1	High priority—a service request will end a DMA transfer.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 PRI = 0
1120 CALL IOPEN(ISC,PRI) 'Set low priority
```

For QuickBASIC:

```
ISC& = 7
PRI% = 0
CALL IOPEN(ISC&,PRI%) 'Set low priority
```

For QBasic:

This command is not supported with QBasic.

Bus Activity

None.

Comments

The BASIC PEN ON, PEN OFF, and PEN STOP statements are unchanged—except that they apply to service requests after IOPEN is executed. In addition, the PEN(1) function returns a value showing whether the service request ended a DMA transfer:

PEN(1)	**Meaning**
0	No DMA transfer was affected by the last service request.
1	DMA transfer was ended by the last service request.

To disable service requests from causing an ON PEN event, use IORESET. The SHELL command in BASIC also disables service request events.

Possible errors are NOERR, ERANGE, and ESEL.

IOPPOLL

This command performs a parallel poll of the interface. It sets a variable to a value (0 to 255) representing the response of those instruments on the interface that respond to a parallel poll.

Syntax

`IOPPOLL (`*select_code*`,`*response*`)`

select_code	specifies the interface select code.
response	variable into which the parallel poll response byte is to be placed. The allowable range is 0 to 255. The eight bits of the response byte correspond to the eight HP-IB data lines (DIO1 through DIO8). Thus, a value of 32 would indicate that some device has responded to the parallel poll with a "1" on DIO6.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 CALL IOPPOLL(ISC,RESPONSE)
```

For QuickBASIC and QBasic:

```
ISC& = 7
CALL IOPPOLL(ISC&,RESPONSE%)
```

Bus Activity

- ATN and EOI are asserted for 25 microseconds.
- The poll byte is read.
- EOI is cleared.
- ATN is restored to its previous state.

Comments

During a parallel poll, each enabled device may put a "1" on an assigned HP-IB data line according to its service request status—otherwise, the line is a "0". There are eight data lines (though more than one device may be assigned to one line). Using a parallel poll, several devices can indicate their service request status simultaneously. The *response* variable contains the state of the eight data lines: DIO1 (in bit 0) through DIO8 (in bit 7).

If the *response* variable contains a "1" in any bit, a device assigned to the corresponding HP-IB line has the service request status the device was set up to report. (See IOPPOLLC.) For example, a device may be set up to report on line DIO4 when it requests service. If an IOPPOLL command shows a "1" in bit 3 of *response*, your program knows the device needs service (assuming no other device is assigned to that line).

Not all devices are capable of responding to a parallel poll. Consult your particular device manuals for specifics.

Possible errors are NOERR, ECTRL, and ESEL.

IOPPOLLC

This command performs a Parallel Poll Configure. In preparation for a parallel poll command, it tells an instrument how to respond affirmatively to the parallel poll, and on which data line to respond.

In general, it sets a parallel poll response byte to reflect the response of a desired arrangement of instruments. Typically, you could define the bits to reflect the responses of particular instruments, or the result of a logical OR operation on several instrument responses.

Refer to IOPPOLL for more information.

Syntax

`IOPPOLLC` (*device_address*, *configuration*)
`IOPPOLLC` (*select_code*, *configuration*)

device_address	specifies the bus address of the device to be configured.
select_code	specifies the interface select code.
configuration	sent to the specified device indicating how it's to respond to a parallel poll. (See "Comments" below.)

Examples

For GW-BASIC:

```
1100 DEVICE = 723
1110 CONF = 10 'Respond with a "1" on line DIO3.
1120 CALL IOPPOLLC(DEVICE,CONF)
```

For QuickBASIC and QBasic:

```
DEVICE& = 723
CONF% = 10 'Respond with a "1" on line DIO3.
CALL IOPPOLLC(DEVICE&,CONF%)
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- PPC is sent.
- PPE is sent.

If a select code is specified:

- PPC is sent.
- PPE is sent.

Comments

The *configuration* parameter defines both the HP-IB line on which to respond and the service request status to indicate. It represents an eight-bit byte described below.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	Response (0 or 1)	DIO Line (DIO1 to DIO8)		

Bit 3 specifies the meaning of an affirmative response. Bits 2 through 0 specify the data line (DIO8 through DIO1). The valid range for *configuration* is 0 to 15—other values cause an error.

Parallel Poll Configuration	Bits	Value
Affirmative response for service request	00001xxx	8
Affirmative response for no service request	00000xxx	0
Respond on line DIO8	0000x111	7
Respond on line DIO7	0000x110	6
Respond on line DIO6	0000x101	5
Respond on line DIO5	0000x100	4
Respond on line DIO4	0000x011	3
Respond on line DIO3	0000x010	2
Respond on line DIO2	0000x001	1
Respond on line DIO1	0000x000	0

For example, to set up a device to indicate an affirmative response ("1") on line DIO5 if it needs service, the configuration value would be 8 + 4 = 12. Alternatively, for the device to indicate an affirmative response ("1") on line DIO5 when it *doesn't* need service, the configuration value would be 0 + 4 = 4.

Not all devices can be configured to respond to a parallel poll. Consult your particular device manuals for specifics.

Possible errors are NOERR, ETIME, ESEL, ECTRL, and ERANGE.

IOPPOLLU

This command performs a Parallel Poll Unconfigure (PPU). It directs an instrument to not respond to a parallel poll. It can be addressed to the interface or a specific device. Refer to IOPPOLLC for more information.

Syntax

IOPPOLLU (*device_address*)
IOPPOLLU (*select_code*)

device_address	specifies a device address.
select_code	specifies the interface select code.

Examples

For GW-BASIC:

```
1100 DEV = 722
1110 CALL IOPPOLLU(DEV)
```

For QuickBASIC and QBasic:

```
DEV& = 722
CALL IOPPOLLU(DEV&)
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- PPC is sent.
- PPD is sent.

If a select code is specified:

- ATN is sent.
- PPU is sent.

Comments

Some devices cannot be unconfigured from the bus. Consult your particular device manuals for specifics.

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

IOREMOTE

This command places a device in Remote mode to disable the device front panel. It can be addressed to the interface or to a specific device.

Syntax

IOREMOTE (*device_address*)
IOREMOTE (*select_code*)

device_address	specifies a device address.
select_code	specifies the interface select code.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 DVM = 723
1120 CALL IOREMOTE(DVM) 'Place the DVM in remote mode.
.
.
1150 CALL IOREMOTE(ISC) 'Set the interface REN line.
```

For QuickBASIC and QBasic:

```
ISC& = 7
DVM& = 723
CALL IOREMOTE(DVM&) 'Place the DVM in remote mode.
   .
   .
CALL IOREMOTE(ISC&) 'Set the interface REN line.
```

Bus Activity

If a device address is specified:

- REN is set.
- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.

If a select code is specified, then REN is set.

Comments

If a select code is specified, a device will not switch into Remote mode until it is addressed to listen.

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

IOREQUEST

This command sets up a serial poll status byte for the HP 82335 and optionally asserts the Service Request (SRQ) line.

Syntax

IOREQUEST (*select_code*, *status*)

select_code	specifies the interface select code.
status	specifies the serial poll status byte. If bit 6 in the status byte is set, the SRQ line will be asserted. If bit 6 is clear, SRQ will not be asserted.

Examples

For GW-BASIC:

```
1100 DEV = 7
1200 STATUS = &H42
1300 CALL IOREQUEST(DEV, STATUS)
1400 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

For QuickBASIC:

```
DEV& = 7
STATUS% = &H42
CALL IOREQUEST(DEV&, STATUS%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

For QBasic:

This command is not supported with QBasic.

Bus Activity

If bit 6 is set in the status parameter: SRQ is asserted.

If bit 6 is clear in the status parameter: no bus activity.

Comments

The HP 82335 interface must not be active controller, or else an ECTRL error will result.

Possible errors are NOERR, ECTRL, and ESEL.

IORESET

This command sets the interface to its start-up state, in which it is not listening and not talking.

In addition, it sets the following HP-IB parameters as indicated:

- The interface timeout is set to 0 seconds (the timeout is disabled).
- The interface EOI mode is enabled.
- High-speed data output is disabled.
- Service-request events are disabled.
- CR/LF is set as the EOL default.
- LF is set as the IOMATCH default.
- If the interface was system controller, then it will also become active controller.

Syntax

IORESET (*select_code*)

select_code	specifies the interface select code.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 CALL IORESET(ISC)
```

For QuickBASIC and QBasic:

```
ISC& = 7
CALL IORESET(ISC&)
```

Bus Activity

If the interface is system controller:

- IFC is pulsed (at least 100 microseconds).
- REN is cleared (at least 100 microseconds).
- ATN is cleared.

Computer Museum

If the interface is non-system controller:

- No bus activity.

Comments

This command returns all devices on the interface to local (front panel) control.

Possible errors are NOERR and ESEL.

IOSEND

This command sends any sequence of user-specified HP-IB commands to the interface. For example, to send an output command to several instruments simultaneously, you can establish their talk/listen status with the IOSEND command, then issue the output command specifying a select code rather than a device address.

Syntax

`IOSEND` (*select_code*, *commands*, *length*)

select_code	specifies the interface select code.
commands	specifies a string of characters, each of which is treated as an interface command.
length	specifies the number of characters in the command string. (An error occurs if the number is less than 0.)

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 COMMANDS$ = "?)/4"
1120 'Specifies unlisten, then listen addresses 9, 15, and 20.
1130 LENGTH = 4
1140 CALL IOSEND(ISC,COMMANDS$,LENGTH)
1150 CALL IOTRIGGER(ISC)
1160 'Triggers devices at addresses 9, 15, and 20.
```

For QuickBASIC and QBasic:

```
ISC& = 7
COMMANDS$ = "?)/4"
'Specifies unlisten, then listen addresses 9, 15, and 20.
LENGTH% = 4
CALL IOSEND(ISC&,COMMANDS$,LENGTH%)
CALL IOTRIGGER(ISC&)
'Triggers devices at addresses 9,15, and 20.
```

Bus Activity

- ATN is set.
- Command bytes are sent.

Comments

See appendix B for a list of HP-IB commands and the corresponding data characters.

All bytes are sent with ATN set. The EOL sequence is not appended, nor is EOI set.

Possible errors are NOERR, ETIME, ESEL, ECTRL, and ERANGE.

IOSPOLL

This command performs a serial poll of a specified device. It sets a variable representing the device's response byte.

Syntax

`IOSPOLL` (*device_address*, *response*)

device_address	specifies the bus address of the device to be polled.
response	variable into which the response byte is placed.

Examples

For GW-BASIC:

```
1100 DEVICE = 723
1110 CALL IOSPOLL(DEVICE,RESPONSE)
1120 'Perform a serial poll on device 723,
1130 'put the response byte in RESPONSE.
```

For QuickBASIC and QBasic:

```
DEVICE& = 723
CALL IOSPOLL(DEVICE&,RESPONSE%)
'Perform a serial poll on device 723,
'put the response byte in RESPONSE%.
```

Bus Activity

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- SPE is sent.
- ATN is cleared.
- Poll byte is read.
- ATN is set.
- SPD is sent.
- UNT is sent.

Comments

If a device is requesting service, it stops requesting service when its response byte is read.

Some devices are not capable of responding to a serial poll, in which case polling may result in an error. Consult the instrument manual to determine if an instrument can respond to a serial poll and how its response byte is interpreted.

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

IOSTATUS

This command determines the current interface status regarding a particular condition. It sets a variable representing that status.

Syntax

`IOSTATUS (`*select_code*`,`*condition*`,`*status*`)`

select_code specifies the interface select code.

condition specifies the condition being checked, from 0 to 11. The possible conditions are:

Value	**Description**
0	Is the interface currently in the remote state? (always no)
1	What is the current state of the SRQ line?
2	What is the current state of the NDAC line?
3	Is the interface currently system controller?
4	Is the interface currently active controller?
5	Is the interface currently addressed as talker?
6	Is the interface currently addressed as listener?
7	What is the interface's bus address?
8	What is the state of the ATN line?
9	What is the address status of the interface?
10	What is on the DIO lines now?
11	What is the bus status of the interface?
12	What interface card is installed?

status variable into which the condition's status is placed. It can have the following values:

Conditions 0 to 6 and 8

Value	Meaning
0	Clear or No
1	Set or Yes

Condition 7

Value	Meaning
0 to 30	Address of card

Condition 9*

Bit	Value	Meaning
0	1	ulpa
1	2	TADS
2	4	LADS
3	8	TPAS
4	16	LPAS
5	32	ATN
6	64	LLO
7	128	REM

Condition 10

Value	Meaning
0 to 255	Value of the data lines on the bus

Condition 11*

Bit	Value	Meaning
0	1	REN
1	2	IFC
2	4	SRQ
3	8	EOI
4	16	NRFD
5	32	NDAC
6	64	DAV
7	128	ATN

* The actual value returned from conditions 9 and 11 will be the sum of the values of all true conditions. For example, the value returned if bits 2 and 3 were true would be 12.

To check whether a specific condition is true, use the AND operand in your language. For example, to check if DAV is true, you could call `IOSTATUS(7L,11,&result)`, then check whether (result AND 32) = 32, then (DAV is set). Make sure you are using the binary AND in your language and not the logical AND.

Condition 12

Value	Meaning
0	no card
1	HP 82990 (old)
2	HP 82335

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 SELECT = 1
1120 CALL IOSTATUS(ISC,SELECT,STATUS)
1130 'Determine if SRQ is set.
```

For QuickBASIC and QBasic:

```
ISC& = 7
SELECT% = 1
CALL IOSTATUS(ISC&,SELECT%,STATUS%)
'Determine if SRQ is set.
```

Bus Activity

None.

Comments

Possible errors are NOERR, ESEL, and ERANGE.

Status conditions 9 through 11 are rarely used.

IOTAKECTL

This command takes active control from the currently active controller back to the HP-IB card.

Syntax

`IOTAKECTL` (*select_code*,*priority*)

select_code	specifies the interface select code.
priority	specifies the priority of the request. This parameter can take one of three values:

1	Wait until the active controller passes control back to me. It will wait until it receives control or until it times out as specified by the IOTIMEOUT function.
2	Assert SRQ with bits 1 and 6 set, then wait until the active controller passes control back to me. It will wait until either it receives control or until it times out as specified by the IOTIMEOUT function.
3	Assert the Interface Clear (IFC) line. Asserting the IFC line immediately makes the HP 82335 the active controller. The HP 82335, however, *must* be the system controller to be able to assert the IFC line. If it is not the system controller, an ECTRL error will result.

Examples

For GW-BASIC:

```
1100 DEV = 7 : PRIORITY = 1
1200 CALL IOTAKECTL(DEV, PRIORITY)
1300 IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

For QuickBASIC:

```
DEV& = 7 : PRIORITY% = 1
CALL IOTAKECTL(DEV&, PRIORITY%)
IF PCIB.ERR <> NOERR THEN ERROR PCIB.BASERR
```

For QBasic:

This command is not supported with QBasic.

Computer Museum

Bus Activity

Bus activity is controlled by the active controller until IOTAKECTL is finished.

Comments

The Command Library defaults to address 30. If you need to, you can change this using the IOCONTROL command.

It may take awhile for the device that has active control to pass control back to the Command Library. You may want to increase your timeout value using IOTIMEOUT before calling IOTAKECTL, and decrease it after the IOTAKECTL call.

Possible errors are NOERR, ETIME, ESEL, ERANGE, and ECTRL.

IOTIMEOUT

This command sets an interface timeout value in seconds for I/O operations that do not complete (for example, the printer runs out of paper).

The default is timeout disabled.

Syntax

IOTIMEOUT (*select_code*, *timeout*)

select_code	specifies the interface select code.
timeout	specifies the length of the timeout period. A value of 0.0 disables the timeout, while a negative value results in an error.

Examples

For GW-BASIC:

```
1100 ISC = 7
1110 TIMEOUT.VAL = 2 'Timeout = 2 seconds.
1120 CALL IOTIMEOUT(ISC,TIMEOUT.VAL)
```

For QuickBASIC and QBasic:

```
ISC& = 7
TIMEOUT.VAL! = 2 'Timeout = 2 seconds.
CALL IOTIMEOUT(ISC&,TIMEOUT.VAL!)
```

Bus Activity

None.

Comments

Timeout is effective for any interface operation that transfers data or commands.

A timeout error occurs only if timeout is enabled (that is, the timeout is set to a positive value).

Timeout should be established early in your program. It provides a way to recover from I/O operations that are not completed.

The timeout value is a real number specified in seconds, which gets rounded to the nearest 1/16 second. To timeout after 5 seconds, set timeout to 5.0. To timeout after 0.5 second, set timeout to 0.5. Note that a timeout of 0.0 effectively disables any timeouts. The maximum allowable timeout is 4096 seconds.

If a transfer times out, the error variable PCIB.ERR returns a value of 100004, which corresponds to the ETIME error, and the error string PCIB.ERR$ returns the message "HPIB: timeout".

Possible errors are NOERR, ESEL, and ERANGE.

IOTRIGGER

This command triggers one or more devices.

Syntax

IOTRIGGER (*device_address*)
IOTRIGGER (*select_code*)

device_address	specifies a device address.
select_code	specifies the interface select code.

Examples

For GW-BASIC:

```
1100 DEV = 723
1110 CALL IOTRIGGER(DEV)
```

For QuickBASIC and QBasic:

```
DEV& = 723
CALL IOTRIGGER(DEV&)
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- GET is sent.

If a select code is specified:

- ATN is set.
- GET is sent.

Comments

Only one device can be triggered at a time if a device address is specified.

If a select code is specified, all devices on the bus that are addressed to listen (with IOSEND, for example) are triggered.

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

5

Pascal Programming

Introduction

This chapter explains how to use the HP-IB Command Library for Pascal programming.

Supported versions of Pascal are listed on the *Supported Languages* sheet included with the Command Library. For example, you can use certain versions of Microsoft Pascal and Borland Turbo Pascal.

This chapter contains several sections describing how you can use the Command Library with Pascal:

- Copying the necessary Library files to a work disk.
- Creating, compiling, and running a Pascal program.
- Processing errors.
- Learning about parameters for Library commands.
- Checking example programs. Two listings at the end of this chapter show how you can use Library commands in Pascal programs.

Detailed syntax information for the commands as they're used with Pascal is included in chapter 7, "Pascal and C Reference."

Copying Files

To begin programming in Pascal, you must copy the Pascal Library files to your work disk.

The HP-IB Command Library disks contain an INSTALL program that copies the Pascal Library files to your system for you.

To use INSTALL:

1. Insert the Library disk into your flexible disk drive—if you're using 5.25-inch disks, use the disk labeled "Disk 1—Install."
2. Run INSTALL by typing

   ```
   a:install (Enter)
   ```

3. Follow the instructions displayed on the screen. When you successfully complete the instructions, the following files are copied:

   - **For Microsoft Pascal:**

     PHPIB.LIB
     IODECL.EX
     IOPROC.EX
     PASCL.BAT
     EXAMPLE.PAS
     MSSCOPE.PAS

   - **For Turbo Pascal:**

     PnnHPIB.TPU *
     TIODECL.EX
     TPASCL.BAT
     TEXAMPLE.PAS
     TMETER.PAS

   * nn represents the Turbo Pascal version number, such as 60 for version 6.0 or 55 for version 5.5.

If you specified invalid drives, or if the system disk is write-protected, no files will be copied. Also make sure you copy all the necessary files from your Pascal compiler and linker to your system. Refer to your Pascal manual for details.

Programming in Pascal

For Pascal programming, the Library is implemented as a series of functions that execute the commands. The functions always return a value indicating the error status of the command.

Writing a Pascal Prggram

You can create a Pascal program using a text editor or the Turbo Pascal integrated environment.

In an application program, you would typically use the commands in the following manner to execute an operation:

- Set up the required variables.
- Perform the operation.
- Test to see if the operation completed successfully.

In the following Microsoft Pascal example, you follow these steps to program two instruments—an HP 3325A Synthesizer/Function Generator and an HP 3456A Digital Voltmeter. You program the source to output a 2-V rms signal, swept from 1 kHz to 10 kHz. You program the DVM to take 20 readings from the signal and output them to an array. Finally, you display the readings on the screen.

Use a convenient text editor to write your program. File names in the example are for Microsoft Pascal—names for Turbo Pascal are given in the comments below.

1. Define some preliminary information.

```
PROGRAM example (input, output);

{$INCLUDE: 'IODECL.EX'}

CONST
 isc = 7;
 source = 717;
 dvm = 722;

TYPE
 str10 = STRING(10);

VAR
 err : INTEGER;

{$INCLUDE: 'IOPROC.EX'}
```

- Include a compiler directive to access IODECL.EX, which declares constants and types. For Turbo Pascal, see the note below.
- For error handling, declare a 10-character string type to hold the name of the command in which an error may occur.
- Declare a variable `err` to represent the return status of subsequent function calls.
- Define an interface select code constant `isc` as 7.
- Define a source address constant `source` as 717.
- Define a voltmeter address constant `dvm` as 722.
- Include a compiler directive to access IOPROC.EX, which declares procedures and functions. For Turbo Pascal, this command isn't used—see the following note.

Note

Be sure to include the compiler directives in their proper places:

- For Microsoft Pascal, `{$INCLUDE: 'IODECL.EX'}` after the program name and `{$INCLUDE: 'IOPROC.EX'}` at the end of the declaration section, before any user-defined procedures or functions that use Library calls.
- For Turbo Pascal, `USES PnnHPIB;` and `{$I TIODECL.EX}` after the program name. (Replace the nn with a 2-digit version number, such as P60HPIB for Turbo Pascal 6.0.)

2. Write an error handling procedure.

```
PROCEDURE Error_handler (error:integer; routine:str10);

BEGIN
 IF error <> noerr THEN BEGIN
   WRITELN ('Error in call to ',routine);
   WRITELN (ERROR:6,Errstr (error));
   WRITE ('Press <Enter> to continue: ');
   READLN;
 END;
END;
```

- In the procedure header, declare an `error` variable to hold the error number, and a `routine` string to hold the name of the command in which the error occurred.
- Declare `estring` to hold the error message string.
- If there is an error (`error <> noerr`):
  - □ Call the HP-IB Library error handling procedure `Errstr`. Pass it the command return variable `err` and `estring` to hold the error message. The `Errstr` routine returns error message strings. It is explained under "Pascal Error Handling" in this chapter.

- Print a message telling where the error occurred.
- Print the error number and the error message string.
- Display a prompt to continue when Enter is pressed.
- Include a READLN statement to accept the Enter. (Ctrl-C terminates the program.)

3. Initialize the bus and the instruments.

```
PROCEDURE Initialize;

BEGIN
 err := IORESET (isc);
 Error_handler (err,'IORESET   ');
 err := IOTIMEOUT (isc, 5.0);
 Error_handler (err,'IOTIMEOUT ');
 err := IOCLEAR (isc);
 Error_handler (err,'IOCLEAR   ');
END;
```

- Set the interface to its default configuration.
- Define a timeout of 5 seconds. Note that the timeout parameter, passed by value, can be expressed as a literal (5.0) in Pascal.
- Return all devices to a known state with IOCLEAR.
- This program calls the `Error_handler` procedure after each command. Pass it the command's return variable (which contains the error number) and the name of the command. Include enough spaces to set the command name field to 10 spaces.

4. Program the source.

```
PROCEDURE source_setup;

VAR
 codes : STRING(38);

BEGIN
 codes := 'RF2 FU1 ST1KH SP10KH MF1KH AM2VR TI5SE';
 err := IOOUTPUTS (source,codes,38);
 Error_handler (err,'IOOUTPUTS ');
END;
```

- Declare the **codes** string to hold instrument programming codes.
- Assign the necessary source programming codes to the **codes** string:

  RF2—select the rear panel signal output.
  FU1—select the sine wave function.
  ST1KH—select a starting frequency of 1 kHz.
  SP10KH—select a stopping frequency of 10 kHz.
  MF1KH—select a marker frequency of 1 kHz.
  AM2VR—select an amplitude of 2 V rms.
  TI5SE—select a sweep time of 5 seconds.
- Send the programming codes to the source with IOOUTPUTS.

5. Program the voltmeter.

```
PROCEDURE dvm_setup;

VAR
 codes: STRING (38);

BEGIN
 codes := 'H SM004 F2 R4 FL0 Z0 4STG 20STN RS1 T4';
 err := IOOUTPUTS (dvm,codes,38);
 Error_handler (err,'IOOUTPUTS ');
END;
```

- Declare the codes string to hold instrument programming codes.
- Assign the necessary programming codes to codes:

  H—software-reset the voltmeter.

  SM004—set the service request mask to enable the voltmeter to set the interface SRQ line when it finishes taking readings (when the Data Ready bit of the response byte is set).

  F2—select the AC volts function.
  R4—select the 10 volt range.
  FL0—turn off filtering.
  Z0—turn off auto zero.
  4STG—select the 4-digit display.
  20STN—take 20 readings.
  RS1—turn on reading storage.
  T4—select hold trigger.
- Send the codes to the voltmeter with IOOUTPUTS.

6. Trigger the instruments.

```
PROCEDURE Trigger;

VAR

 codes : STRING(2);

BEGIN
 err := IOTRIGGER (dvm);
 Error_handler (err,'IOTRIGGER ');
 codes := 'SS';
 err := IOOUTPUTS (source,codes,2);
 Error_handler (err,'IOOUTPUTS ');
END;
```

- Declare the **codes** string to hold instrument programming codes.
- Use IOTRIGGER to trigger the voltmeter.
- Enter the programming code necessary to trigger the source into the **codes** string. Send it (with the proper length parameter) to the source with IOOUTPUTS.

These lines demonstrate that some instruments respond to an HP-IB trigger command, while others must be triggered with instrument-specific programming codes.

7. Wait for the voltmeter to finish reading.

```
PROCEDURE Wait_for_SRQ;

CONST
 srqline = 1;

VAR
 response : INTEGER;
 done : BOOLEAN;

BEGIN
 done := FALSE;
 REPEAT
  REPEAT
   err := IOSTATUS (isc,srqline,response);
   Error_handler (err,'IOSTATUS  ');
  UNTIL response <> 0;
  err := IOSPOLL (dvm,response);
  Error_handler (err,'IOSPOLL   ');
  done := ((response AND 68) = 68);
 UNTIL done;
END;
```

- Assign the interface condition whose status is being checked to `srqline`. In this case, check for condition 1—the SRQ line.
- Declare `response` and `done` variables for use in status checking.
- Set `done` to false before the status check.
- Use IOSTATUS with `srqline` set to 1 to see if the interface SRQ line has been set. Put the result in `response`.
- As long as `response` is zero, repeat the status check because the voltmeter has not yet set the SRQ line (indicating it is finished). As soon as `response` changes to some nonzero value, perform a serial poll of the voltmeter to see

which of its conditions, if any, set the SRQ line. Also, the serial poll clears the SRQ.

- The result of the serial poll is the status byte of the voltmeter returned in `response`. Compare `response` and the value 68—the sum of the SRQ bit (64) and the Data Ready bit (4). If these bits are set, **done** is true and the program continues. If they are not set, **done** is false and the status check is performed again.

8. Enter the readings into an array and print them.

```
PROCEDURE Readout;

VAR
 codes : STRING (14);
 length : INTEGER;
 readings : REALARRAY(20);
 i : INTEGER;
 numvalues : INTEGER;

BEGIN
 numvalues := 20;
 length := 14;
 codes := 'S01 -20STR RER';
 err := IOOUTPUTS (dvm,codes,length);
 Error_handler (err,'IOOUTPUTS ');
 err := IOEOI (isc,0);
 Error_handler (err,'IOEOI     ');
 err := IOENTERA (dvm,readings,numvalues);
 Error_handler (err,'IOENTERA  ');
 WRITELN ('The readings are: ');
 FOR i := 1 TO numvalues DO WRITELN (readings[I]);
END;
```

- Declare all necessary variables: `codes` to hold instrument programming codes; `length` to define the length of the codes; `readings` to hold the voltmeter readings; **i** for a for/next loop; and **numvalues** for the number of values to enter. For Turbo Pascal, **readings** should be defined as a regular ARRAY.

- Program the voltmeter to output its stored readings:

  SO1—turn on system output mode.
  –20STR—unstore 20 readings.
  RER—recall (output) the 20 readings.

- Disable the EOI mode so reading doesn't terminate after entering only one value.
- Set the number of readings to 20 and use IOENTERA to enter the readings from the voltmeter. Put them in the **readings** array.
- Use a loop to print the readings from the **readings** array.

9. Assemble the procedures for execution.

```
BEGIN {main}
 Initialize;
 Source_setup;
 Dvm_setup;
 Trigger;
 Wait_for_SRQ;
 Readout;
END.
```

Saving the Pascal Program

When you have finished writing the program, save it as PROGRAM.PAS in the Library directory LIB. (The method by which you save it depends on your text editor.)

Compiling and Linking with Microsoft Pascal

After the Pascal program is finished, you must compile it and link it. This example assumes that the necessary compiling and linking programs are on disk C, as is the Library directory. See the Microsoft Pascal manual for more details.

Note These instructions may differ for different revisions of Microsoft Pascal.

From MS-DOS, select the Library directory as the current directory. For example

```
c: (Enter)
cd \hpib (Enter)
```

You can compile and link a Microsoft Pascal program in two ways:

- Automatically using a Command Library program.
- Separate compile and link steps using Microsoft Pascal commands.

Automatic Compiling and Linking

The batch file PASCL.BAT in your Command Library automatically compiles and links a specified Microsoft Pascal program. (You should make sure your PATH environment variable includes the Command Library directory and the Pascal executables directory.)

To use PASCL.BAT, type at the MS-DOS prompt (where *filename* is the name of your program without the .PAS extension)

```
pascl filename (Enter)
```

For example, to compile and link the above program, type

```
pascl program (Enter)
```

Compiling and Linking Separately

From MS-DOS, start pass one of the compiler. You can use the default compilation by typing (where *filename* is the name of the program without the .PAS extension)

```
pas1 filename; [Enter]
```

Alternatively, you'll be prompted for compiler options if you type

```
pas1 [Enter]
```

For this example, type

```
pas1 program; [Enter]
```

If there are no errors, `Pass One No errors detected.` is displayed.

To run pass two, type

```
pas2 [Enter]
```

Pass two displays no intermediate prompts on the screen. If there are no errors, pass two displays the following when it is complete:

```
Code Area Size =
Cons Area Size =
Data Area Size =

Pass Two  No Errors Detected.
```

Note Pass three is not required for this example.

When compilation is successfully completed, link the Microsoft Pascal program. You can use the default compilation by typing (where *filename* is the name of the program without the .PAS or .OBJ extension)

```
link filename,,,phpib; [Enter]
```

Alternatively, you'll be prompted for linker options if you type

```
link [Enter]
```

For this example, type

```
link program,,,phpib; Enter
```

This directs the linker to link the Microsoft Pascal Command Library file PHPIB.LIB to your program. It also links the Microsoft Pascal system library file and all necessary modules in the run-time library. The MS-DOS prompt is displayed when linking is complete.

Compiling with Turbo Pascal

After the Pascal program is finished, you must compile it. (No separate linking process is required—compiling and linking occur together.) This example assumes that the necessary compiling programs are on drive C, as is the Library directory. See the Turbo Pascal manual for more details.

Note These instructions may differ for different revisions of Turbo Pascal.

From MS-DOS, select the Library directory as the current directory. For example

```
c: Enter
cd \hpib Enter
```

You can compile a Turbo Pascal program in three ways:

- Automatically using a Command Library program.
- Using the Turbo Pascal command-line compiler.
- By running it from the Turbo Pascal integrated environment.

Using the Command Library Program

The batch file TPASCL.BAT in your Command Library automatically compiles a specified Turbo Pascal program. (You should make sure your PATH environment variable includes the Command Library directory and the Pascal executables directory.)

To use TPASCL.BAT, type at the MS-DOS prompt (where *filename* is the name of your program without the .PAS extension)

`tpascl` *filename* (Enter)

For example, to compile the above program, type

`tpascl program` (Enter)

Using the Command-Line Compiler

From MS-DOS, type at the MS-DOS prompt (where *filename* is the name of your program without the .PAS extension)

`tpc` *filename* (Enter)

Compiling in the Integrated Environment

You can compile and run a program from the Turbo Pascal integrated environment by selecting those choices from the menus in the integrated environment.

Running the Pascal Program

When you are ready to run the program, connect the SIGNAL output of the source to the VOLTS input of the DVM. (Include a 50-ohm load in this line to ensure proper readings.) Use HP-IB cables to connect the instruments to your computer. The result of the compiling and linking procedure is an executable program file called PROGRAM.EXE, which is saved in the current directory. When you are ready to run it, type

`program` (Enter)

Watch the display on the function generator. You will see the various functions (sine wave, AC volts, sweep time) displayed as they are programmed. The voltmeter displays its operation as well—you can watch it take readings, store them, and output them to the "Readings" array. As the program ends, it displays the readings on your screen.

Pascal Error Handling

General information about Command Library errors and how to process errors are contained in "Processing I/O Errors" in chapter 1 and in appendix A, "Error Descriptions."

If you don't need to write a special error-processing routine for your program, you can write one like the following:

```
PROCEDURE Checkerror (error : integer; routine : str12);

VAR
 estr: xxstr40;

BEGIN
 IF error <> noerr THEN BEGIN
  estr := errstr (error);
  WRITELN ('Error ',error:4,estr,' was encountered');
  WRITELN (' in call to HP-IB function ',routine);
  WRITELN;
  WRITELN ('Press <Enter> to continue: ');
  READLN;
 END;
END;
```

This example uses "errstr"—a function provided in the IOPROC.EX or TIODECL.EX file that returns an error message string corresponding to the error value.

The parameter `routine` in Checkerror gives you an indication of where the error occurred.

`noerr` is one of the mnemonic constants established in the IODECL.EX or TIODECL.EX file that correspond to the possible Command Library errors—other error mnemonics are listed under "Command Library Errors" in chapter 1.

Checkerror might then be used in your program as follows:

```
  :
err := IOENTER (709,reading);
Checkerror (err,'IOENTER   ');
  :
```

Command Library Parameters

This section presents information about Command Library parameters as they are used with Pascal.

Passing Parameters

In Pascal, you can pass parameters to procedures and functions in two ways: pass by value and pass by reference. Those passed by reference are indicated by *_REF* attached to their designators in the syntax reference later in this manual.

With pass by value, a copy of the current value of the parameter is made for the called routine, and the original copy of the data is unchanged. In this case, you may specify a variable (such as `isc`), a literal (such as 709), or an expression (such as, `isc * 100 + 9`) as the parameter in the call statement.

With pass by reference, only a single copy of the data exists. Therefore, any changes made to the variable in the subroutine are reflected in both the called routine and the calling procedure. Whenever a reference parameter is indicated, you must use a variable—literals and expressions are not allowed.

Parameter Types

Several types of variables are used to describe parameters to Library command calls in chapter 7, "Pascal and C Reference."

Integer Expression

An integer expression is any valid combination of integers, integer variables, integer functions, and integer operators that evaluate to a single integer value.

The valid range for this variable type is integers from −32,768 to 32,767. Integers are used to specify flags and other discrete information. For example

```
701
isc
isc * 100 + 9
700 + ord('$')
```

Long/Four-Byte Integer Expression

A long integer (Turbo Pascal) or four-byte integer (Microsoft Pascal) is similar to an integer expression, except the valid range is from −2,147,483,648 through 2,147,483,647. Anywhere a long/four-byte integer expression is indicated, you can use an integer expression instead. Long/four-byte integers are used to specify select codes and device addresses, including extended addresses.

Integer Variable

An integer variable is a variable of type INTEGER that is to be passed by reference. You may not use long/four-byte variables when an integer variable is specified, nor may you use a constant, literal, or integer expression. Integer variables are used for returning meaningful integer values, such as with IOGETTERM or IOSPOLL.

Real Expression

A real expression is any valid combination of real numbers, real variables, real functions, and real number operators that evaluates to one real value. The range of real numbers is limited to approximately 2×10^{-39} to 2×10^{38} (negative and positive) and the precision to seven digits. (Turbo Pascal six-byte real numbers are converted to four-byte numbers internally, so the range is as stated here.) They're used to specify numeric data. For example

```
1.23
data_value
sqrt(data_value)
2 * pi * r
```

Real Variable

A real variable is a variable declared to be of type REAL that is to be passed by reference. You may not use constants, literals, or other expressions. They're used to return numeric data from input transfers.

Real Array

A real array is an array of real variables. When you declare arrays for Microsoft Pascal, use the super type REALARRAY—it's defined in the IODECL.EX file. For Turbo Pascal, use arrays of type REAL. Although the theoretical size limit for arrays is 32,767 elements, the actual limit is smaller because of memory size limitations. They're used for numeric data.

Be careful when using very large arrays. Declared globally, they can result in link-time errors. Declared within subroutines, they may compile and link without error but cause system problems later when printed or accessed in computations.

Each of the following examples allocates an array of 20 elements, which can be accessed as `values[1]` through `values[20]`. Each element is a real number.

- **For Microsoft Pascal:**

```
VAR values: REALARRAY(20);

TYPE real20 = REALARRAY(20);
VAR values: real20;
```

- **For Turbo Pascal:**

```
VAR values: ARRAY[1..20] of REAL;

TYPE real20 = ARRAY[1..20] of REAL;
VAR values: real20;
```

String Variable

A string variable is a variable declared to be of type STRING or LSTRING that is to be passed by reference. You must establish the string size before you call a Library function that uses the string variable. Valid string sizes are 1 through 32,767. You may not use string literals (such as `'My String'`). Be careful when you use very large strings. String variables are used to specify characters and other ASCII text.

Each of the following examples declares a string with size 10.

```
VAR instring: STRING(10);

TYPE str10 = STRING(10);
VAR instring: STR10;
```

Character

A character is any variable of type CHAR or a constant, literal, or character function that represents a single ASCII character. A variable with type STRING(1) is not compatible with a CHAR variable. In the example above, `instring[1]` is a character. For example

```
ch
'A'
chr(10)
```

Any Type of Array

Any type of array can be a real array, an integer array, a double/eight-byte real array, a string, or other type of array. It indicates the place to start reading or storing data.

Each of the following examples allocates an array. (For Microsoft Pascal, use the four array types defined for binary transfers by the IODECL.EX file: BININT, BINREAL, BINDOUBLE, and BINCHAR.)

- **For Microsoft Pascal:**

```
VAR readings : BINDOUBLE(50);

TYPE int20 = BININT(20);
VAR values: int20;

TYPE str10 = STRING(10);
VAR instring: str10;
```

- **For Turbo Pascal:**

```
VAR readings : ARRAY[1..50] of REAL8;

TYPE int20 = ARRAY[1..20] of INTEGER;
VAR values: int20;

TYPE str10 = STRING(10);
VAR instring: str10;
```

Example Programs

Oscilloscope Example

The following program is written in Microsoft Pascal. The program uses two devices: HP 54601A digitizing oscilloscope (or compatible scope) and a printer capable of printing HP Raster Graphics Standard, such as a ThinkJet printer.

The program tells the scope to take a reading on channel 1 and send the data back to this program. Then it prints some simple statistics about the data. 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 HP-IB transaction.

Things to note about this program:

- Note the use of the IOENTERAB command. This command will read an arbitrary block of data as defined in IEEE-488.2. IOENTERAB can read either definite length or indefinite length arbitrary block data.
- If your instrument sends data in some other block data format, you can use the IOENTERB and IOOUTPUTB commands in conjunction with IOENTERS and IOOUTPUTS, respectively, to simulate these other formats.
- You should probably disable character matching before executing an IOENTERB or IOENTERAB because the character in the "match" string is generally a valid binary value, rather than a termination character.

- The commands that are sent to the scope are device dependent and are found in the manual for the scope.
- The error checking in the program consists of checking the return value of each Command Library function after it is called. In this program, a procedure (`error_handle`) has been set up to check this value automatically.
- Before an IOENTERS statement is executed, space should be allocated for the string (if it's an LSTRING) by assigning the string to '   ', where the number of spaces is the maximum length to be entered. If this is not done, the string will be of length 0, and no characters will be entered. You can then shorten the string to the correct length (in case less than the maximum number of characters were entered) by using the DELETE procedure.

The program has three main parts to it:

Computer Museum

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

```
(*
  This program tells the scope to take a reading on channel 1, then
  sends the data back to this program.  We can do anything we want
  to the data at this time, and we choose to print some simple
  statistics about the data.  The program then tells the scope to send
  the data directly to the printer, illustrating how the controller
  doesn't have to be directly involved in an HP-IB transaction.
*)

program hpib_sample(input,output);

{$include: 'iodecl.ex'}

const
    max_str_len = 200;
type
    strtype = lstring(max_str_len);
var
    isc,
    scope      : integer4;
    cmd         : strtype;
    pre         : realarray(10);
    reason      : integer;
    status      : integer;
    length      : integer;
    bytes       : integer;
    readings    : binint(5000);

{$include: 'ioproc.ex'}

procedure error_handle(error : integer; routine: strtype);
begin
    if error <> NOERR then begin
        writeln('Error in call to ', routine, error:3, errstr(error));
        abort('HPIB error', 1, 0);
    end;
end;

procedure sendcmd (cmd: strtype);
var
    length : integer;
begin
    length := retype(integer, cmd.len);
    error_handle (IOOUTPUTS (scope, cmd, length), 'IOOUTPUTS') ;
end;

procedure initialize;
begin
    isc := 7;
    scope := isc * 100 + 7;

    (* initialize the hpib interface and scope
     *)
    error_handle(IORESET(isc), 'IORESET');
    error_handle(IOTIMEOUT(isc, 5.0), 'IOTIMEOUT');
    error_handle(IOCLEAR(scope), 'IOCLEAR');
    error_handle(IOREMOTE(isc), 'IOREMOTE');
end;

procedure get_data;
var
```

```
    i : integer;
begin

    (* setup scope to accept waveform data
     *)
    sendcmd('*RST') ;
    sendcmd (':autoscale') ;

    (* setup up the waveform source
     *)
    sendcmd (':waveform:format word') ;

    (* input waveform preamble to controller
     *)
    sendcmd (':digitize channel1') ;
    sendcmd (':waveform:preamble?') ;

    length := 10 ;
    error_handle (IOENTERA (scope, pre, length), 'IOENTERA') ;

    (* turn off 'lf' enter terminator and turn on EOI
     * This is required, as 'lf' is a valid binary value.
     *)
    error_handle(IOMATCH(isc, chr(10), 1), 'IOMATCH');
    error_handle(IOEOI(isc, 1), 'IOEOI');

    (* command scope to send data
     *)
    sendcmd (':waveform:data?') ;

    (* enter the data
     *)
    bytes := 8000;
    error_handle(IOENTERAB(scope, readings, bytes, 2), 'IOENTERAB');

    (* use IOGETTERM to see if all points were entered
     *)
    error_handle(IOGETTERM(isc, reason), 'IOGETTERM');

    if (reason and 1) = 0 then
        writeln('Not all points received');
    (* Read the last byte from the scope.  This must always be done
     * after an IOENTERAB command.  If the character read is a
     * 'lf', then the device is done sending data.  If the character
     * read is a ';' or a ',', then the device is waiting to send
     * another block of data.
     *
     * Note also that we can use the select code instead of the device
     * address for the first parameter of this command.  This is because
     * the scope is still addressed to talk, and the computer to listen
     * from the IOENTERAB command.
     *)
    cmd := ' ' ;
    length := 1 ;
    error_handle (IOENTERS (isc, cmd, length), 'IOENTERS #1') ;

    (* cmd[1] is the first character of string
     *)
    if (cmd [1] <> chr(10)) then
    begin
        writeln ('scope wants to send more data...') ;
    end;
end;

procedure massage_data;
var
    vdiv  : real ;
    off   : real ;
    sdiv  : real ;
    delay : real ;
    i     : integer ;

begin
    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
     *)
    sendcmd ('*IDN?') ;
    cmd := NULL ;
    for i := 1 to max_str_len do
        concat (cmd, ' ');
    length := max_str_len ;
    error_handle (IOENTERS (scope, cmd, length), 'IOENTERS #2') ;
    delete (cmd, length, upper(cmd) - length);

    (*  print the statistics about the data
     *)
    writeln ('Oscilloscope ID: ', cmd) ;
    writeln (' ----------  Current settings  -----------') ;
    writeln ('      Volts/Div = ',vdiv  , ' V') ;
    writeln ('         Offset = ',off   , ' V') ;
    writeln ('          S/Div = ',sdiv  , ' S') ;
    writeln ('          Delay = ',delay , ' S') ;
end;

procedure print_data;
begin

    (* tell the scope to SRQ on 'operation complete'
     *)
    sendcmd ('*SRE 32 ; *ESE 1') ;

    (* tell the scope to print
     *)
    sendcmd (':print? ; *OPC') ;

    (* 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 := chr(63) * chr(32+1) * chr(64+7) ;

    (* 63 is unlisten *)
    (* printer is at address 1, add 32 to get listen cmd *)
    (* scope is at address 7, add 64 to get talk command *)

    length := retype(integer, cmd.len);
    error_handle(IOSEND(isc, cmd, length), 'IOSEND');

    (* now, the ATN line must be set to FALSE.
     *)
    error_handle(IOCONTROL(isc, 8, 0), 'IOCONTROL');

    (* wait for SRQ before continuing program
     *)
    status := 0 ;
    while status = 0 do begin
        while status = 0 do begin
            (* stay in this while loop until SRQ is asserted *)
            error_handle(IOSTATUS(isc, 1, status), 'IOSTATUS');
        end;

        (* make sure it was the scope requesting service
         *)
        error_handle(IOSPOLL(scope, status), 'IOSPOLL');
        (* 64 = bit 6 set *)
        status := status and 64 ;
    end;
end;

procedure cleanup;
begin

    (* give local control back to the scope
     *)
    error_handle(IOLOCAL(isc), 'IOLOCAL');

end;

begin { main }
    initialize;
    get_data;
    massage_data;
    print_data;
    cleanup;
end.
```

Multimeter Example

```
{
 This example uses the HP 34401A Multimeter as the primary device.
 We will also use the HP 3325A Function Generator as a source for
 the multimeter.

 This example sets up the meter to take 128 readings, reads the data
 into an array, then plots the data on the screen.  In effect, it
 turns the multimeter into a simple oscilloscope.  This program is
 also checking other devices that are on the bus to see if they need
 service.  The SRQ line along with parallel and serial polling is
 used to make these checks.  The program will continue until the user
 presses a key on the PC keyboard.
}

program main(input, output);

uses Graph, Crt, p60hpib;

{$I tiodecl.ex}

const
    NUM_READINGS = 128 ;

type   (* type declarations follow *)
    strtype = string[255];
    arrtype = array [1..NUM_READINGS] of real;

var
    isc           :longint;
    dvm           :longint;
    source        :longint;
    device_addr_1 :longint;
    device_addr_2 :longint;
    cmd           :strtype;
    len           :integer;
    response      :integer;
    readings      :arrtype;
```

```
procedure cleanup; forward;

procedure error_handle(error : integer; routine: strtype);
var retval : INTEGER ;
begin
    if error <> NOERR then begin

        (* we have an error, so let's abort all activity on the HPIB bus
         *)
        retval := IOABORT(isc) ;

        cleanup;
        writeln('Error in call to ', routine, error:3, errstr(error));
        halt(1);
    end;
end;

procedure cleanup;
var retval : INTEGER ;
begin

    (* clear the dvm so we can send the commands to reset it
     *)
    retval := IOCLEAR(dvm) ;

    (* reset the dvm
     *)
    cmd := ':DISP:STATE ON; *RST' ;
    len := length(cmd);
    retval := IOOUTPUTS(dvm, cmd, len) ;

    (* unconfigure the parallel poll
     *)
    retval := IOPPOLLU(isc) ;

    (* restore video mode
     *)
    CloseGraph;
end;

procedure get_data;
```

```
var
    i       : integer;
    ymin,
    ymax    : real;
    xaxis,
    yaxis   : integer;
    temp    : real;
    textout : string;
begin

    (*  Ask the DVM to send us the data
     *)

    str (NUM_READINGS:0, cmd) ;
    cmd := ':SAMPLE:COUNT ' + cmd + '; :READ?' ;
    len := length(cmd);
    error_handle(IOOUTPUTS(dvm, cmd, len), 'IOOUTPUTS #2');

    (*  Read the data
     *)

    len := NUM_READINGS ;
    error_handle(IOENTERA(dvm, readings, len), 'IOENTERA #1');


    (*  graph the data
     *)

    (* clear screen, and draw a border around the screen
     *)
    ClearDevice;
    MoveTo(0,0);
    LineTo(GetMaxX,0);
    LineTo(GetMaxX,GetMaxY);
    LineTo(0,GetMaxY);
    LineTo(0,0);

    (* find the minimum and maximum values in the data
     *)
    ymin := readings[1];
    ymax := readings[1];
```

```
for i:=1 to len do begin
    if (readings[i] < ymin) then ymin := readings[i];
    if (readings[i] > ymax) then ymax := readings[i];
end;

(* print some labels
 *)
str(ymax:0, textout);
textout := 'MAX = ' + textout;
OutTextXY(2, 2, textout);

str(ymin:0, textout);
textout := 'MIN = ' + textout;
OutTextXY(2, GetMaxY-TextHeight(textout), textout);

(* scale the min and max values to give extra space on top & bottom
 * of screen
 *)
if ymin > 0 then
    ymin := ymin *  0.6
else
    ymin := ymin * 1.4;

if ymax > 0 then
    ymax := ymax * 1.4
else
    ymax := ymax * 0.6;

(* plot the data
 *)
temp := readings[1];
temp := ((temp-ymin) * GetMaxY) / (ymax-ymin);
yaxis := round(GetMaxY-temp);
MoveTo(0, yaxis);
for i := 2 to len do begin
    temp := i;
    temp := temp * GetMaxX;
    temp := temp / (len - 1);
    xaxis := round(temp);
```

```
        temp := readings[i];
        temp := ((temp-ymin) * GetMaxY) / (ymax-ymin);
        yaxis := round(GetMaxY-temp);

        LineTo(xaxis, yaxis);
    end;
end;

procedure poll_device (dev_addr: longint) ;
var
    response : integer;
begin

    (* do a serial poll of the device configured to use parallel
     *  poll line 0
     *)
    error_handle(IOSPOLL(dev_addr, response), 'IOSPOLL #3');

    (* should check RESPONSE here to see if any action needs to be taken.
     * the values that RESPONSE can take are device dependent.
     *)
end;


procedure check_srq;
var
    response : integer;
begin
    (* conduct a parallel poll
     * note that the source doesn't respond to parallel poll's,
     *  so we need to poll that device separately.
     *)
    error_handle(IOPPOLL(isc, response), 'IOPPOLL #1');
    if ((response and 1) <> 0) then
        poll_device (device_addr_1);

    if ((response and 2) <> 0) then
        poll_device (device_addr_2);

    (* check all devices that were configured to respond to
     *  parallel poll
     *)

    (* check any other devices on the bus here that weren't
     *  configured to respond to parallel poll by performing
     *  a serial poll on each one.
     *)
    error_handle(IOSPOLL(source, response), 'IOSPOLL #2');

    (* see if we've cleared the srq yet
     *)
    error_handle(IOSTATUS(isc, 1, response), 'IOSTATUS #3');
    if (response = 1) then begin
        cleanup;
        writeln('SRQ locked high');
        halt(1);
    end;
end;

procedure setup;
begin
    (* let's use dma to send the strings to program the devices
     *)
    error_handle(IODMA(isc, 40, 3), 'IODMA #1');

    (* program the function generator
     *)
    cmd := 'RF1 FR30HZ FU1 ST1KH SP10KH MF1KH AM1VR TI5SE' ;
    len := length(cmd);
    error_handle(IOOUTPUTS(source, cmd, len), 'IOOUTPUTS #1');

    (* program the dvm
     *)
    cmd := ':CONF:VOLT:DC 30,.1;';
    cmd := cmd + ':ZERO:AUTO OFF;';
    cmd := cmd + ':TRIG:DELAY MIN;';
    cmd := cmd + ':DISP:STATE OFF;';
    cmd := cmd + '';
    len := length(cmd);
    error_handle(IOOUTPUTS(dvm, cmd, len), 'IOOUTPUTS #2');

    (* turn dma off again
     *)
    error_handle(IODMA(isc, 0, 3), 'IODMA #2');
end;

procedure initialize;
var
    GraphDriver : integer;
    GraphMode   : integer;
    ErrCode     : integer;
    TempChar    : char ;
begin
    isc := 7;
    dvm := isc * 100 + 22;
    source := isc * 100 + 12;
    device_addr_1 := isc * 100 + 20;
    device_addr_2 := isc * 100 + 7;

    (* initialize the hpib interface and scope
     *)
    error_handle(IORESET(isc), 'IORESET');
    error_handle(IOTIMEOUT(isc, 3.0), 'IOTIMEOUT');
    error_handle(IOCLEAR(source), 'IOCLEAR #1');
    error_handle(IOCLEAR(dvm), 'IOCLEAR #2');
    error_handle(IOFASTOUT(isc, 1), 'IOFASTOUT');
    TempChar := chr(10);
    error_handle(IOEOL(isc, TempChar, 0), 'IOEOL');

    (* We will now configure all devices that can respond to a parallel
     *  poll.  This example assumes devices at addresses 20 and 7 can
     *  respond to a parallel poll.  see operators manual of individual
     *  devices to see if they can respond to a parallel poll.
     *)

    (* configure the device at address 20 for a parallel poll
     *)
    error_handle(IOPPOLLC(device_addr_1, $08), 'IOPPOLLC #1');

    (* configure the device at address 7 for a parallel poll
     *)
    error_handle(IOPPOLLC(device_addr_2, $09), 'IOPPOLLC #2');
    (* configure any other devices that can respond to parallel poll here
     *)

    (* set video mode
     *)
    GraphDriver := Detect;
    InitGraph(GraphDriver, GraphMode, '');
    ErrCode := GraphResult;
    if ErrCode <> grOk then
    begin
        writeln ('Graphics error:', GraphErrorMsg(ErrCode));
    end;

end;

begin  { main }

    initialize;

    setup;

    while not keypressed do
    begin
        error_handle(IOSTATUS(isc, 1, response), 'IOSTATUS #1');
        if response = 1 then
            check_srq;
        get_data ;
    end;

    cleanup;
end.
```

C Programming

Introduction

This chapter explains how to use the HP-IB Command Library with the C programming language.

Supported versions of C are listed on the *Supported Languages* sheet included with the Command Library. For example, you can use certain versions of Microsoft C, Microsoft QuickC, and Borland Turbo C. This chapter contains several sections describing how you can use the Command Library with C:

- Using various memory models.
- Copying the necessary Library files to a work disk.
- Creating, compiling, and running a C program.
- Processing errors.
- Learning about parameters for Library commands.
- Checking example programs. Two listings at the end of this chapter show how you can use Library commands in C programs.

Detailed syntax information for the commands as they're used with C is included in chapter 7, "Pascal and C Reference."

Specifying Memory Models

When used with C, the HP-IB Command Library uses the "large" memory model. You should ensure that your C program is compatible with the Command Library by using one of these methods:

- Use the large memory model for your program. However, not all C languages let you use the large memory model.
- Prototype all Command Library functions as "far" calls in your program and all parameters passed by reference as "far" references—enabling the program to use a smaller memory model. The Command Library includes the CFUNC.H file, which makes these declarations.

Consult your C manual for a discussion of the differences between memory models.

Copying Files

To begin programming in C, you must copy the C Command Library files to your work disk.

The HP-IB Command Library disks contain an INSTALL program that copies the C Library files to your system for you.

To use INSTALL:

1. Insert the Library disk into your flexible disk drive—if you're using 5.25-inch disks, use the disk labeled "Disk 1—Install."
2. Run INSTALL by typing

   ```
   a:install (Enter)
   ```

3. Follow the instructions displayed on the screen. When you successfully complete the instructions, the following files are copied:

   - **For Microsoft C and QuickC:**

     CLHPIB.LIB
     QCHPIB.QLB
     CHPIB.H

CFUNC.H
MSCL.BAT
QCCL.BAT
EXAMPLE.C
MSMETER.C

- **For Turbo C:**

  CLHPIB.LIB
  CHPIB.H
  CFUNC.H
  TCCL.BAT
  EXAMPLE.C
  TSCOPE.C
  TCHHPIB.C

If you specified invalid drives, or if the system disk is write-protected, no files will be copied. Also make sure you copy all the necessary files from your C compiler and linker to your system. Refer to your C manual for details.

Programming in C

For C programming, the Library is implemented as a series of function calls. The functions always return a value indicating the error status of the command.

Writing a C Program

You can create a C program using a text editor or the QuickC or Turbo C integrated environment.

In an application program, you typically use the commands in the following manner:

1. Set up the required variables.
2. Perform the operation.
3. Test to see if the operation completed successfully.

In the following example, you follow these steps to program two instruments—an HP 3325A Synthesizer/Function Generator and an HP 3456A Digital Voltmeter. You program the source to output a 2-V rms signal, swept from 1 kHz to 10 kHz. You program the DVM to take 20 readings from the signal and output them to an array. Finally, you display the readings on the screen.

Use a convenient text editor to write your program. File names in the example are for Microsoft C and QuickC—names for Turbo C are given in the comments below each example.

1. Define some preliminary information.

```
#include "CHPIB.H"
#include "CFUNC.H"

#define  ISC       7L
#define  SOURCE  717L
#define  DVM     722L
#define  SRQLINE   1
int      error;
```

- Include a compiler directive to access CHPIB.H and CFUNC.H, which define some constants, an error string procedure, and function prototyping. CFUNC.H isn't mandatory for a large memory model program, although its inclusion helps reveal variable-type problems. If you are compiling C++ programs with Turbo C++, instead of including the files CFUNC.H and CHPIB.H, you need to add the following statements to your program:

  ```
  extern "C"
  {
     #include <cfunc.h>
     #include <chpib.h>
  }
  ```

  Note that this does not apply when you are compiling standard C programs in Turbo C++; only C++ programs using C++ constructs require this.
- Define an interface select code constant `ISC` as 7.
- Define a source address constant `SOURCE` as 717.
- Define a voltmeter address constant `DVM` as 722. This example assumes select code 7, voltmeter address 22, and source address 17. Also, `ISC`, `SOURCE`, and `DVM` must all be LONG values.
- Define the constant `SRQLINE` for use as the condition parameter in calls to IOSTATUS.

- Declare a variable `error` to represent the return status of subsequent function calls.

2. Write an error handling procedure.

```
error_handler (error, routine)
int      error;
char     *routine;

{
  char     ch;

  if (error != NOERR)
    {
     printf ("Error in call to %s \n", routine);
     printf ("%d %s \n", error, errstr(error));
     printf ("Press Enter to continue: ");
     scanf ("%c", &ch);
    }
}
```

- In the procedure header, declare an `error` variable to hold the error number, and a `routine` string to hold the name of the command in which the error occurred.
- If there is an error (`error != NOERR`):
  - □ Print a message telling where the error occurred.
  - □ Print the error number and the error message string. (The `errstr` routine returns error message strings. It is explained in the "C Error Handling" section of this chapter.)
  - □ Display a prompt to continue when Enter is pressed.
  - □ Include a `scanf` statement to accept the Enter. (Ctrl-C terminates the program.)

3. Initialize the bus and the instruments.

```
initialize ()
{
 error = IORESET (ISC);
 error_handler (error, "IORESET");
 error = IOTIMEOUT (ISC, 5.0);
 error_handler (error, "IOTIMEOUT");
 error = IOCLEAR (ISC);
 error_handler (error, "IOCLEAR");
}
```

- Set the interface to its default configuration.
- Define a timeout of 5 seconds. Note that the timeout parameter, passed by value, can be expressed as a literal (5.0) in C. Be sure to include the decimal point—otherwise, the C compiler will assume you are passing an integer value even though a double value is required.
- Return all devices to a known state with IOCLEAR.
- Call `error_handler` after each command. Pass it the command's return variable (which contains the error number) and the name of the command. You can combine these operations—for example `error_handler (IORESET (ISC),"IORESET");`

4. Program the source.

```
sourcesetup ()
{
  char     *codes;
  codes =  "RF2 FU1 ST1KH SP10KH MF1KH AM2VR TI5SE";
  error =  IOOUTPUTS (SOURCE, codes, 38);
  error_handler (error, "IOOUTPUTS ");
}
```

- Declare the **codes** string to hold instrument programming codes.
- Assign the necessary source programming codes to the **codes** string:

  RF2—select the rear panel signal output.
  FU1—select the sine wave function.
  ST1KH—select a starting frequency of 1 kHz.
  SP10KH—select a stopping frequency of 10 kHz.
  MF1KH—select a marker frequency of 1 kHz.
  AM2VR—select an amplitude of 2 V rms.
  TI5SE—select a sweep time of 5 seconds.

- Send the programming codes to the source with IOOUTPUTS.

5. Program the voltmeter.

```
dvmsetup ()
{
  char    *codes;
  codes = "H SM004 F2 R4 FL0 Z0 4STG 20STN RS1 T4";
  error = IOOUTPUTS (DVM, codes, 38);
  error_handler (error, "IOOUTPUTS");
}
```

- Declare the **codes** string to hold instrument programming codes.
- Assign the necessary programming codes to **codes**:

  H—software-reset the voltmeter.

  SM004—set the service request mask to enable the voltmeter to set the interface SRQ line when it finishes taking readings (when the Data Ready bit of the serial poll response byte is set).

  F2—select the AC volts function.
  R4—select the 10 volt range.
  FL0—turn off filtering.
  Z0—turn off auto zero.
  4STG—select the 4-digit display.

20STN—take 20 readings.
RS1—turn on reading storage.
T4—select trigger hold.

- Send the codes to the voltmeter with IOOUTPUTS.

6. Trigger the instruments.

```
trigger ()
{
  error = IOTRIGGER (DVM);
  error_handler (error, "IOTRIGGER");
  error = IOOUTPUTS (SOURCE, "SS", 2);
  error_handler (error, "IOOUTPUTS");
}
```

- Use IOTRIGGER to trigger the voltmeter.
- Send the programming code necessary to trigger the source ("SS") using the proper length parameter (2) with IOOUTPUTS. Note that a literal string can be used as a parameter to IOOUTPUTS.

These lines demonstrate that some instruments respond to an HP-IB trigger command, while others must be triggered with instrument-specific programming codes.

7. Wait for the voltmeter to finish reading.

```
wait_for_srq ()
{
  int     response;
  do
    {
     do
       {
         error = IOSTATUS (ISC, SRQLINE, &response);
         error_handler (error, "IOSTATUS");
       }
     while (response == 0);
     error = IOSPOLL (DVM, &response);
     error_handler (error, "IOSPOLL");
    }
  while ((response & 68) != 68);
}
```

- Declare **response** variable for use in status checking.
- Use IOSTATUS with SRQLINE set to 1 to see if the interface SRQ line has been set. Put the result in **response**.
- As long as **response** is zero, repeat the status check because the voltmeter has not yet set the SRQ line (indicating it is finished). As soon as **response** changes to some nonzero value, perform a serial poll of the voltmeter to see which of its conditions, if any, set the SRQ line. Also, the serial poll clears the SRQ. Note that "&" must precede **response** in order for the poll value to be properly returned.
- The result of the serial poll is the status byte of the voltmeter returned in **response**. Compare **response** with the value 68—the sum of the SRQ bit (64) and the Data Ready bit (4). If these bits are set, exit the function—otherwise, the status check is performed again.

8. Enter the readings into an array and print them.

```
readout ()
{
  char      *codes;
  float     readings[20];
  int       i;
  int       numvalues;

  numvalues = 20;
  codes = "SO1 -20STR RER";
  error = IOOUTPUTS (DVM, codes, 14);
  error_handler (error, "IOOUTPUTS");
  error = IOEOI (ISC, 0);
  error_handler (error, "IOEOI");
  error = IOENTERA (DVM, readings, &numvalues);
  error_handler (error, "IOENTERA");
  printf ("\n The readings are: \n");
  for (i = 0; i < numvalues; i++)
      printf ("%f \n", readings[i]);
}
```

Computer Museum

- Declare all necessary variables: `codes` to hold instrument programming codes; `readings` to hold the voltmeter readings; `i` for a for/next loop; and `numvalues` for the number of values to enter.
- Program the voltmeter to output its stored readings:

  SO1—turn on system output mode.
  −20STR—unstore 20 readings.
  RER—recall (output) the 20 readings.
- Disable the EOI mode so reading doesn't terminate after entering only one value.
- Set the number of readings to 20, and use IOENTERA to enter the readings from the voltmeter. Put them in the `readings` array. `readings` does not

need the "&" before it because C always passes a pointer for an array in a function call.

- Use a loop to print the readings from the **readings** array. Remember that C array indices begin at 0.

9. Assemble the procedures for execution in the MAIN function.

```
main ()
{
  initialize ();
  sourcesetup ();
  dvmsetup ();
  trigger ();
  waitforsrq ();
  readout ();
}
```

Saving the C Program

When you have finished writing the program, save it as PROGRAM.C. It may be convenient to save it in the Command Library directory (such as LIB). (The method by which you save it depends on your text editor.)

Compiling and Linking the C Program

After the C program is finished, you must compile and link it. This example assumes that the necessary compiling and linking programs are on disk C, along with the Library directory.

From MS-DOS, select the Command Library directory as the current directory. For example

```
c: Enter
cd \hpib Enter
```

You can compile a C program in three ways:

- Automatically using a Command Library program.
- Using the C command-line compiler.
- By running it from the C integrated environment.

Using the Command Library Program

A batch file in your Command Library automatically compiles a specified C program. The file you use depends upon your C language. (You should make sure your PATH environment variable includes the Command Library directory and the C executables directory.) To compile and link a program, type the following at the MS-DOS prompt (where *filename* is the name of your program without the .C extension)

- Microsoft C (uses file MSCL.BAT)

  `mscl` *progname* (Enter)

- Microsoft QuickC (uses file QCCL.BAT)

  `qccl` *progname* (Enter)

- Turbo C (uses file TCCL.BAT)

  `tccl` *progname* (Enter)

Note Each of these programs uses the large memory model.

For example, to compile the above program using Microsoft C, type

```
mscl program
```
(Enter)

Using the Command-Line Compiler

The compiler command depends upon your C language. From MS-DOS, type the following at the MS-DOS prompt (where *filename* is the name of your program without the .C extension)

- Microsoft C

  `cl /AL` *progname* `clhpib.lib` (Enter)

- Microsoft QuickC

  `qcl /AL` *progname* `clhpib.lib` (Enter)

- Turbo C

  `tcc -ml` *progname* `clhpib.lib` (Enter)

- Borland C++

  `bcc -ml` *progname* `clhpib.lib` (Enter)

Note

Each of these commands specifies the large memory model. If you include the CFUNC.H file in your program, you can omit the memory-model option (`/AL` or `-ml`) from the command to use the default model or specify another memory model.

For example, to compile the example program:

```
cl /AL program clhpib.lib (Enter)
```

Note

Turbo C thinks that the library CLHPIB.LIB is a replacement for its run-time library, CL.LIB. Therefore, if you are compiling Turbo C programs from the Turbo C environment, you need to explicitly list CL.LIB (or appropriate file, depending on what memory model you are using) in your .PRJ files so that both are linked into your program.

If you are compiling Turbo C huge model programs with greater than 64K of static data, be sure to use the library named TCHHPIB.LIB instead of CLHPIB.LIB. This file is located in the directory you specified during installation.

Compiling in the Integrated Environment

You can compile and run a program from the QuickC or Turbo C integrated environment by selecting those choices from the menus. However, you must previously specify CLHPIB.LIB as a required library file from the menu.

Note

If you're using QuickC 1.0, you can't create executable files on disk from the integrated environment because it ignores the Command Library, but you can create them in memory. However, you can create executable files on disk using the Command Library program or the command-line compiler.

Running the C Program

When you are ready to run the program, connect the SIGNAL output of the source to the VOLTS input of the DVM. (Include a 50-ohm load in this line to ensure proper readings.) Use HP-IB cables to connect the instruments to your computer. The result of the compilation and linking procedures is an executable program file called PROGRAM.EXE, which is saved in the current directory. When you are ready to run it, type

```
program (Enter)
```

Watch the display on the function generator. You will see the various functions (sine wave, AC volts, sweep time) displayed as they are programmed. The voltmeter displays its operation as well—you can watch it take readings, store them, and output them to the `readings` array. As the program ends, it displays the readings on your screen.

Using Microsoft FORTRAN With the C Library

Microsoft languages can generally call the Microsoft C versions of functions in the HP-IB Command Library by following the mixed language programming guide included with language. For example, the documentation for Microsoft FORTRAN includes a section titled "FORTRAN Calls to C".

Here is an example using Microsoft FORTRAN version 4.1. It was compiled and linked with the command line `FL HPIB.FOR CLHPIB.LIB`, which compiles the source file HPIB.FOR and links it with the standard FORTRAN library and the library CLHPIB.LIB.

```
C    HPIB.FOR 82335A FORTRAN Example Program
C
C Declare HP-IB functions the program will use
C
C Note that IORESET is a function returning a 2-byte integer,
C  and has one 4-byte integer parameter, ISC.
C

      INTERFACE TO INTEGER*2 FUNCTION IORESET
     +    [C,ALIAS:'_IORESET'] (ISC)
      INTEGER*4 ISC
      END

C Note that the timeout value, N, is an 8-byte real.
C
      INTERFACE TO INTEGER*2 FUNCTION IOTIMEOUT
     +    [C,ALIAS:'_IOTIMEOUT'] (ISC, N)
      INTEGER*4 ISC
      REAL*8 N
      END

C Note that parameters that are pointers to data are declared
C    FAR and REFERENCE
C
      INTERFACE TO INTEGER*2 FUNCTION IOENTERS
     +    [C,ALIAS:'_IOENTERS'] (ISC, S, L)
      INTEGER*4 ISC
      CHARACTER*20 S [FAR, REFERENCE]
      INTEGER*2 L [FAR, REFERENCE]
      END

      INTERFACE TO INTEGER*2 FUNCTION IOOUTPUTS
     +    [C,ALIAS:'_IOOUTPUTS'] (ISC, S, L)
      INTEGER*4 ISC
      CHARACTER*20 S [FAR, REFERENCE]
      INTEGER*2 L
      END
C
C     MAIN START
C
C FORTRAN requires functions to be declared here as well.
      INTEGER*2 IORESET
      INTEGER*2 IOTIMEOUT
      INTEGER*2 IOENTERS
      INTEGER*2 IOOUTPUTS


      INTEGER*4 ISC
      INTEGER*4 DEV_ADDR
      INTEGER*2 ERRNO
      CHARACTER*20 STR
      INTEGER*2 LENGTH

      ISC = 7
      DEV_ADDR = 702
      STR = 'HELLO WORLD'
C
C     CALL IORESET(7)
C
      ERRNO = IORESET(ISC)
      IF (ERRNO .GT. 0) GOTO 100
C
C     CALL IOTIMEOUT(7, 5.0)
C
      ERRNO = IOTIMEOUT(ISC, 5.0)
      IF (ERRNO .GT. 0) GOTO 100
C
C     CALL IOENTERS(702, STR, 10)
C
      LENGTH = 10
      ERRNO = IOENTERS(DEV_ADDR, STR, LENGTH)
      IF (ERRNO .GT. 0) GOTO 100
C
C     CALL IOOUTPUTS(702, 'HELLO WORLD', 11)
C
      LENGTH = 11
      ERRNO = IOOUTPUTS(DEV_ADDR, STR, LENGTH)
      IF (ERRNO .GT. 0) GOTO 100
      GOTO 200

100   CONTINUE
      WRITE (*,*) 'HP-IB ERROR', ERRNO

200   CONTINUE

      END
```

C Error Handling

General information about Command Library errors and how to process errors are contained in "Processing I/O Errors" in chapter 1 and in appendix A, "Error Descriptions."

It's good practice to check for errors after each Command Library call. If you don't need to write a special error-processing routine for your program, you can use the error_handler routine provided in EXAMPLE.C, or you can write one like the following:

```
checkerror (error, routine)

int      error;
char     *routine;

{
char     ch;

if (error != NOERR)

  { printf ("\n Error %d %s \n", error, errstr(error));
    printf (" in call to HP-IB function %s \n\n", routine);
    printf ("Press Enter to continue: ");
    scanf ("%c", &ch);
  }
}
```

This example uses "errstr", a function contained in CHPIB.H that returns an error message string corresponding to the error value.

The parameter routine in CHECKERROR gives you an indication of which Library command produced the error.

NOERR is one of the mnemonic constants established in the file CHPIB.H that correspond to the possible Command Library errors—other error mnemonics are listed under "Command Library Errors" in chapter 1.

Checkerror might then be used in your program as follows:

```
error = IOENTER (709L, &reading);
checkerror (error, "IOENTER");
```

Command Library Parameters

This section presents information about Command Library parameters as they are used with the C language.

Passing Parameters

In C, all single-valued function parameters are passed by value. With pass by value, a copy of the parameter is made for use by the called routine, and the original data is unchanged. If the parameter only provides information to a function, you may specify a variable (such as `isc`), a literal (such as 709), or an expression (such as `isc * 100 + 9`) as the parameter in the function call.

Array and structure variables are passed by far reference. When such a multicomponent variable appears in a parameter list, the address of its first element is used. Any changes made to the data structure in a called function are reflected back in the calling function.

When single-value data must be communicated back from a called function (such as the *reading* value of an IOENTER), pass by value can be overridden. This is done by adding the "address of" operator (&) to the front of the variable name in the parameter list.

Parameters passed by reference are indicated by *_REF* attached to their designators in the syntax reference. This applies to arrays and structures as well as scalar values with the prefix operator "&".

Parameter Types

Several types of variables are used to describe parameters to Library command calls in chapter 7, "Pascal and C Reference."

Integer Expression

An integer expression is any valid combination of integers, integer variables, integer functions, and integer operators that evaluate to a single (two-byte) integer value. The range of integers is limited to whole numbers from −32,768 to 32,767. Integers are used to specify flags and other discrete information. For example (assuming the declaration `int numval;`)

```
32
numval
numval%3
```

Long-Integer Expression

A long-integer expression is similar to an integer expression, except the range of valid values is −2,147,483,648 through 2,147,483,647 (it evaluates to a four-byte value). Integer expressions and long-integer expressions are *not* compatible in parameter lists. Long-integer expressions are used to specify select codes and device addresses, including extended addresses. For example (assuming the declaration `long isc;`)

Computer
Museum

```
701L
isc
isc * 100 + 9
```

Integer Variable

An integer variable is a variable declared to be of type INT (two bytes) that is to be passed by reference (such as by using the "address of" operator "&"). You may not use a long-integer variable when an integer variable is specified, nor may you use a constant, literal, or expression. Integer variables are used for returning meaningful integer values, such as with IOGETTERM or IOSPOLL.

Float Expression

A float expression is any valid combination of real numbers, real variables, real functions, and real number operators that evaluates to a single real value. Only four-byte real numbers are allowed, limiting the range of real numbers from approximately 2×10^{-38} to 2×10^{38} (negative or positive) and the precision to seven digits. Long float and double variables are not compatible with float expressions. Float expressions are used to specify numeric data. For example (assuming the declaration `float data_value, pi, r;`)

```
1.23
data_value
2.0 * pi * r
```

Float Variable

A float variable is a variable of type FLOAT (four bytes) that is to be passed by reference, using the "address of" operator "&". If a float variable parameter is indicated, you may not use constants, literals, or other expressions. Float variables are used to return numeric data from input transfers.

Double Expression

A double expression is any valid combination of real numbers, real variables, real functions, and real number operators that evaluates to a single real value. Only eight-byte real numbers are allowed, giving a range of real numbers from approximately 2×10^{-308} to 2×10^{308} (negative or positive). Float variables are not compatible with double expressions. Double expressions are used to specify numeric data for IOOUTPUT and IOTIMEOUT. For example (assuming the declaration `double timeout;`)

```
timeout
60.0 * timeout
```

Float Array

A float array is an array of FLOAT values. Although the theoretical size limit for arrays is 32,767 elements, the actual limit may be smaller, depending on which memory model you are using. For further details, see your C manual. Be careful not to violate memory restrictions when you use very large arrays. Declared globally, they can result in link time errors. Declared within subroutines, they may compile and link without error, but cause system problems such as stack overflow during execution. Large arrays should be declared outside of functions to save stack space.

String Variable

A string variable is a variable declared to hold multiple characters. It can be either a pointer to a character or a character array. Use caution when using very large strings because linkage or stack problems may occur. Literals (such as `"My String"`) may not be used. For example

```
char *info;
char inbytes[10]
```

String Expression

A string expression may be a string variable or a literal string such as `"My String"`.

Character Expression

A character expression is any single CHAR variable, constant, expression, literal, or character function that represents a single ASCII character.

Any Type of Array

Any type of array can be a float array, an integer array, a double array, a string, or other type of array. It indicates the place to start reading or storing data.

Example Programs

Oscilloscope Example

The following program is written in Turbo C. The program uses two devices: an HP 54601A digitizing oscilloscope (or compatible scope) and a printer capable of printing HP Raster Graphics Standard, such as a ThinkJet printer.

The program tells the scope to take a reading on channel 1 and send the data back to this program. Then it prints some simple statistics about the data. 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 HP-IB transaction. Things to note about this program:

- Note the use of the IOENTERAB command. This command will read an arbitrary block of data as defined in IEEE-488.2. IOENTERAB can read either definite length or indefinite length arbitrary block data.
- If your instrument sends data in some other block data format, you can use the IOENTERB and IOOUTPUTB commands in conjunction with IOENTERS and IOOUTPUTS, respectively, to simulate these other formats.

- You should probably disable character matching before executing an IOENTERB or IOENTERAB because the character in the "match" string is generally a valid binary value, rather than a termination character.
- The commands that are sent to the scope are device dependent and are found in the manual for the scope.
- The error checking in the program consists of checking the return value of each Command Library function after it is called. In this program, a procedure (`error_handle`) has been set up to check this value automatically.

The program has three main parts to it:

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

```
/*
  This program tells the scope to take a reading on channel 1, then
  sends the data back to this program.  We can do anything we want
  to the data at this time, and we choose to print some simple
  statistics about the data.  The program then tells the scope to send
  the data directly to the printer, illustrating how the controller
  doesn't have to be directly involved in an HP-IB transaction.
*/

#include <stdio.h>   /* used for printf ()                 */
#include <stdlib.h>  /* used for exit ()                   */
#include "CHPIB.H"   /* HPIB library constant declarations */
#include "CFUNC.H"   /* HPIB library function prototypes   */

/* function prototypes */
void error_handle (int, char *) ;
void initialize (void) ;
void get_data (void) ;
void massage_data (void) ;
void print_data (void) ;
void cleanup (void) ;
void sendcmd (char *) ;

/* global data */
```

```
long    isc ;
long    scope ;
char    cmd [50] ;
float   pre [10] ;
int     reason ;
int     status ;
int     length ;
int     bytes ;
int     readings [5000] ;

void main ()
{
    initialize () ;
    get_data () ;
    massage_data () ;
    print_data () ;
    cleanup () ;
}

void initialize ()
{
    isc = 7L ;
    scope = isc * 100L + 7L ;

    /* initialize the hpib interface and scope
     */
    error_handle (IORESET (isc), "IORESET") ;
    error_handle (IOTIMEOUT (isc, 5.0), "IOTIMEOUT") ;
    error_handle (IOCLEAR (scope), "IOCLEAR") ;
    error_handle (IOLLOCKOUT (isc), "IOLLOCKOUT") ;
}

void get_data ()
{
    /* setup scope to accept waveform data
     */
    sendcmd ("*RST") ;
    sendcmd (":autoscale") ;

    /* setup up the waveform source
     */
```

```
sendcmd (":waveform:format word") ;

/* input waveform preamble to controller
 */
sendcmd (":digitize channel1") ;
sendcmd (":waveform:preamble?") ;

length = 10 ;
error_handle (IOENTERA (scope, pre, &length), "IOENTERA") ;

/* turn off 'lf' enter terminator and turn on EOI
 * This is required, as 'lf' is a valid binary value.
 */
error_handle (IOMATCH (isc, '\n', 1), "IOMATCH") ;
error_handle (IOEOI (isc, 1), "IOEOI #1") ;

/* command scope to send data
 */
sendcmd (":waveform:data?") ;

/* enter the data
 */
bytes = 8000 ;
error_handle (IOENTERAB (scope, readings, &bytes, 2), "IOENTERAB") ;

/* use IOGETTERM to see if all points were entered
 */
error_handle (IOGETTERM (isc, &reason), "IOGETTERM") ;
if ((reason & 1) == 0)
{
    printf ("Not all points received\n") ;
}

/* Read the last byte from the scope.  This must always be done
 * after an IOENTERAB command.  If the character read is a
 * 'lf', then the device is done sending data.  If the character
 * read is a ';' or a ',', then the device is waiting to send
 * another block of data.
 *
 * Note also that we can use the select code instead of the device
 * address for the first parameter of this command.  This is because
```

```
     * the scope is still addressed to talk, and the computer to listen
     * from the IOENTERAB command.
     */
    length = 1 ;
    error_handle (IOENTERS (isc, cmd, &length), "IOENTERS") ;
    if (cmd [0] != '\n')
    {
        printf ("scope wants to send more data...\n") ;
    }
}

void massage_data ()
{
    float vdiv ;
    float off ;
    float sdiv ;
    float delay ;

    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
     */
    sendcmd ("*IDN?") ;
    length = 49 ;
    error_handle (IOENTERS (scope, cmd, &length), "IOENTERS") ;

    /*  print the statistics about the data
     */
    printf ("\nOscilloscope ID:  %s\n", cmd) ;
    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 ()
{
```

```
/* tell the scope to SRQ on 'operation complete'
 */
sendcmd ("*SRE 32 ; *ESE 1") ;

/* tell the scope to print
 */
sendcmd (":print? ; *OPC") ;

/* 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                        */

length = strlen (cmd) ;
error_handle (IOSEND (isc, cmd, length), "IOSEND") ;

/* now, the ATN line must be set to FALSE.
 */
error_handle (IOCONTROL (isc, 8, 0), "IOCONTROL") ;

/* wait for SRQ before continuing program
 */
status = 0 ;
while (status == 0)
{
    while (status == 0)
    {
        error_handle (IOSTATUS (isc, 1, &status), "IOSTATUS") ;
    }

    /* make sure it was the scope requesting service
     */
    error_handle (IOSPOLL (scope, &status), "IOSPOLL") ;
    status &= 64 ;
}
```

```
    /* clear the status byte so the scope can assert SRQ again
     * if needed.
     */
    sendcmd ("*CLS") ;
}

void cleanup ()
{
    /* give local control back to the scope
     */
    error_handle (IOLOCAL (scope), "IOLOCAL") ;
}


void error_handle (int error, char *routine)
{
    if (error != NOERR)
    {
        printf ("HPIB error in call to %s: %d, %s\n",
                 routine, error, errstr (error)) ;
        exit (1) ;
    }
    return ;
}



void sendcmd (char *cmd)
{
    error_handle (IOOUTPUTS (scope, cmd, strlen (cmd)), "IOOUTPUTS") ;
}
```

Multimeter Example

```
/*
 * This example uses the HP 34401A Multimeter as the primary device.
 * We will also use the HP 3325A Function Generator as a source for
 * the multimeter.
 *
 * This example sets up the meter to take 128 readings, reads the data
 * into an array, then plots the data on the screen.  In effect, it
 * turns the multimeter into a simple oscilloscope.  This program is
 * also checking other devices that are on the bus to see if they need
 * service.  The SRQ line along with parallel and serial polling is
 * used to make these checks.  The program will continue until the user
 * presses a key on the PC keyboard.
 */

#include <string.h>  /* used for strcpy() and strcat() */
#include <graph.h>   /* used for graphics routines */
#include <stdio.h>   /* used for printf() */
#include <stdlib.h>  /* used for exit() */
#include <conio.h>   /* used for kbhit() */
#include "CHPIB.H"   /* HPIB cmd library constant declarations */
#include "CFUNC.H"   /* HPIB cmd library function prototypes */

#define NUM_POINTS 128

/* function prototypes */
void initialize (void) ;
void setup (void) ;
void do_srq (void) ;
void get_data (void) ;
void poll_device (long) ;
void cleanup (void) ;
int check_srq (void) ;
void error_handle (int, char *) ;

/* global data */
long    isc ;
long    dvm ;
long    source ;
long    device_addr_1 ;
long    device_addr_2 ;
char    cmd [200] ;
int     length ;
float   readings [NUM_POINTS] ;

void main ()
```

```
{
    initialize () ;
    setup () ;
    while (!kbhit ())
    {
        /* program can do other work here
         */
        if (check_srq ())
        {
            do_srq () ;
        }
        get_data () ;
    }
    cleanup () ;
}

void initialize ()
{
    isc = 7L ;
    dvm = isc * 100L + 22L ;
    source = isc * 100L + 12L ;
    device_addr_1 = isc * 100L + 20L ;
    device_addr_2 = isc * 100L + 7L ;

    /* initialize the hpib interface and clear devices
     */
    error_handle (IORESET (isc), "IORESET") ;
    error_handle (IOTIMEOUT (isc, 5.0), "IOTIMEOUT") ;
    error_handle (IOCLEAR (source), "IOCLEAR #1") ;
    error_handle (IOCLEAR (dvm), "IOCLEAR #2") ;
    error_handle (IOFASTOUT (isc, 1), "IOFASTOUT") ;
    error_handle (IOEOL (isc, "", 0), "IOEOL") ;

    /* we will now configure all devices that can respond to a parallel
     *  poll this example assumes devices at addresses 20 and 7 can
     *  respond to a parallel poll.  see operators manual of individual
     *  devices to see if they can respond to a parallel poll.
     */

    /* configure the device at address 20 for a parallel poll
     */
    error_handle (IOPPOLLC (device_addr_1, 0x08), "IOPPOLLC #1") ;

    /* configure the device at address 7 for a parallel poll
     */
    error_handle (IOPPOLLC (device_addr_2, 0x09), "IOPPOLLC #2") ;

    /* configure any other devices that can respond to parallel poll here
     */

    /* set the video mode
     */
    _setvideomode (_HRESBW) ;
}

void setup ()
{
    /* let's use dma to send the strings to program the devices
     */
    error_handle (IODMA (isc, 40, 3), "IODMA #1") ;

    /* program the function generator
     */
    strcpy (cmd, "RF1 FR30HZ FU1 ST1KH SP10KH MF1KH AM1VR TI5SE") ;
    length = strlen (cmd) ;
    error_handle (IOOUTPUTS (source, cmd, length), "IOOUTPUTS #1") ;

    /* program the dvm
     */

    strcpy (cmd, ":CONF:VOLT:DC 30,.1 ;") ;
    strcat (cmd, ":ZERO:AUTO OFF ;") ;
    strcat (cmd, ":TRIG:DELAY MIN ;") ;
    strcat (cmd, ":DISP:STATE OFF ;") ;

    length = strlen (cmd) ;
    error_handle (IOOUTPUTS (dvm, cmd, length), "IOOUTPUTS #2") ;

    /* turn dma off again
     */
    error_handle (IODMA (isc, 0, 3), "IODMA #2") ;
}
```

```
void do_srq ()
{
    int response ;

    /* conduct a parallel poll
     * note that the source doesn't respond to parallel poll's, so
     *  we need to serial poll that device separately.
     */
    error_handle (IOPPOLL (isc, &response), "IOPPOLL #1") ;
    if ( (response & 1) != 0)
        poll_device (device_addr_1) ;

    if ( (response & 2) != 0)
        poll_device (device_addr_2) ;

    /* check all devices that were configured to respond to
     *  parallel poll
     */

    /* check any other devices on the bus here that weren't
     *  configured to respond to parallel poll by performing
     *  a serial poll on each one.
     */
    error_handle (IOSPOLL (source, &response), "IOSPOLL #2") ;

    /* see if we've cleared the srq yet
     */
    error_handle (IOSTATUS (isc, 1, &response), "IOSTATUS #3") ;
    if (response == 1)
    {
        printf ("SRQ locked high\n") ;
        cleanup () ;
        exit (1) ;
    }
}

void get_data ()
{
    int     i ;
    float   ymin, ymax ;
```

```
long    xaxis, yaxis ;
char    buffer [80] ;

/* Ask the DVM to send us the data
 */
sprintf (cmd, ":SAMPLE:COUNT %d ;", NUM_POINTS) ;
strcat (cmd, ":READ?") ;
length = strlen (cmd) ;
error_handle (IOOUTPUTS (dvm, cmd, length), "IOOUTPUTS #2") ;

/* Read in the data
 */
length = NUM_POINTS ;
error_handle (IOENTERA (dvm, readings, &length), "IOENTERA") ;

/*  graph the data
 */

/* set mode, clear screen, and draw a border around the screen
 */
_clearscreen (_GCLEARSCREEN) ;
_moveto (0,0) ;
_lineto (639,0) ;
_lineto (639,199) ;
_lineto (0,199) ;
_lineto (0,0) ;

/* find the minimum and maximum values in the data
 */
ymin = readings [0] ;
ymax = readings [0] ;

for (i=0 ; i < length ; i++)
{
    if (readings [i] < ymin) ymin = readings [i] ;
    if (readings [i] > ymax) ymax = readings [i] ;
}

/* print some labels
 */
_settextposition (2,2) ;
```

```
    sprintf (buffer, "MAX = %f", ymax) ;
    _outtext (buffer) ;

    _settextposition (24,2) ;
    sprintf (buffer, "MIN = %f", ymin) ;
    _outtext (buffer) ;

    /* scale the min and max values to give extra space on top & bottom
     * of screen
     */
    if (ymin > 0.0)
        ymin = ymin * 0.6 ;
    else
        ymin = ymin * 1.4 ;

    if (ymax > 0)
        ymax = ymax * 1.4 ;
    else
        ymax = ymax * 0.6 ;

    /* plot the data
     */
    _moveto (0, (short) (200 - (readings [0] - ymin) * 200 / (ymax - ymin))) ;
    for (i=0 ; i < length ; i++)
    {
        xaxis = (long)i*640L/ ( (long)length-1L) ;
        yaxis = 200.0 - (readings [i] - ymin) * 200.0 / (ymax - ymin) ;
        _lineto ( (short) xaxis, (short) yaxis) ;
    }
}

void poll_device (long device_addr)
{
    int response ;

    /* do a serial poll of the device
     */
    error_handle (IOSPOLL (device_addr, &response), "IOSPOLL #3") ;

    /* should check RESPONSE here to see if any action needs to be taken.
     * the values that RESPONSE can take are device dependen
     */
}

void cleanup ()
{
    /* clear the dvm so we can send the commands to reset it
     */
    IOCLEAR (dvm), "IOCLEAR#4" ;

    /* reset the dvm
     */
    strcpy (cmd, ":DISP:STATE ON ;") ;
    strcat (cmd, "*RST") ;
    length = strlen (cmd) ;
    IOOUTPUTS (dvm, cmd, length), "IOOUTPUTS#5" ;

    /* unconfigure the parallel poll
     */
    IOPPOLLU (isc), "IOPPOLLU" ;

    /*  set video mode back to normal
     */
    _setvideomode (_DEFAULTMODE) ;
}

int check_srq ()
{
    int response ;

    error_handle (IOSTATUS (isc, 1, &response), "IOSTATUS #1") ;
    return response ;
}

void error_handle (int error, char *routine)
{
    if (error != NOERR)
    {
        /* we have an error, so let's abort all activity on the HPIB bus
         */
        error_handle (IOABORT (isc), "IOABORT") ;
        cleanup () ;
        printf ("HPIB error in call to %s: %d, %s\n",
                routine, error, errstr (error)) ;
        exit (1) ;
    }
    return ;
}
```

7

Pascal and C Reference

This chapter presents a detailed Command Library syntax reference for Pascal and C languages.

Parameters for Library commands are separated into several groups according to the types of arguments you must provide. The following table summarizes these groups. See "Command Library Parameters" in chapters 5 and 6 for more detail about parameter types for Pascal and C.

Parameter Type	Pascal	C
Select Codes and Addresses	Long/four-byte integer expression	Long-integer expression
Flags and Discrete Information*	Integer variable or expression	Integer variable or expression
Numeric Data (Single)*	Real variable or expression	Float variable (for IOENTER)—double variable or expression (for IOOUTPUT and IOTIMEOUT)
Numeric Data (Array)*	Real array	Float array
Binary Data (Array)*	Any type of array	Any type of array
String Data*	String variable	String variable or expression
Character Data	Character	Character expression

* For parameters marked *_REF*, a variable or array must be passed by reference.

You can use literals and expressions for simple parameters that provide information to the command—but *not* for parameters that return information.

Parameters that must be passed by reference are indicated by *_REF* attached to their designators in this chapter. For C, all array variables must be passed by far reference.

Throughout this chapter, HP-IB terms are listed by abbreviation rather than by name. For example, "Go To Local" is listed as "GTL." A complete list of HP-IB abbreviations is included in appendix B, "Summary of HP-IB."

IOABORT

This command aborts all activity on the interface. IOABORT will ab
as much as it can depending upon its current system controller and active controller status.

Syntax

`IOABORT` (*select_code*)

select_code	specifies the interface select code.

Computer Museum

Examples

For Pascal:

```
error  : INTEGER ;

error  := IOABORT(7)  (* for Microsoft Pascal *)
if error <> NOERR then writeln('an error occurred...');
```

For C:

```
int  error  ;

error = IOABORT(7L)
if (error != NOERR) printf ("an error occurred...\n");
```

Bus Activity

If the HP 82335 is system controller:

- IFC is pulsed at least 100 microseconds.
- REN is set.
- ATN is cleared.

If the HP 82335 is active, but not system controller:

- UNT is sent.

If the HP 82335 is neither active nor system controller:

- No bus activity.

Comments

Devices in Local Lockout will remain locked out.

Possible errors are NOERR and ESEL.

If the HP 82335 was the system, but not active controller, IOABORT will make the HP 82335 both system and active controller.

IOCLEAR

This command returns a device to a known, device-dependent state. It can be addressed to the interface or to a specific device.

Syntax

`IOCLEAR` (*device_address*)
`IOCLEAR` (*select_code*)

device_address	specifies the address of a device to be cleared.
select_code	specifies the select code of the interface on which all devices are to be cleared.

Examples

For Pascal:

```
VAR
 err : INTEGER;

err := IOCLEAR (723); {Clear the device at address 23.}

err := IOCLEAR (7);   {Clear all devices on the interface.}
```

For C:

```
int     error;

error = IOCLEAR(723L); /*Clear the device at address 23.*/
    .
    .
error = IOCLEAR(7L); /*Clear all devices on the interface.*/
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- SDC is sent.

If a select code is specified:

- ATN is set.
- DCL is sent.

Comments

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

IOCONTROL

This command directly sets status conditions in the interface. It can be used to address or unaddress the interface as a talker or listener, or set the interface's bus address. IOCONTROL can also change system controller status of the HP 82335 interface.

Note IOCONTROL should be used with caution since it operates directly on the interface.

Syntax

`IOCONTROL` (*select_code*, *condition*, *status*)

select_code	specifies the interface select code.
condition	specifies the status condition that is to be set. Conditions which can be set are:

Value	Description
3	Make the interface the non-system or system controller.
5	Address or unaddress the interface as talker.
6	Address or unaddress the interface as listener.
7	Set the interface's bus address.
8	Clear or set ATN.

status variable into which the condition's status is placed. It can have the following values:

Condition 3

Value	Meaning
0	Make interface non-system controller
1	Make interface system controller

Conditions 5 and 6

Value	Meaning
0	Clear this condition
1	Set specified condition

Condition 7

Value	Meaning
0 to 30	Bus address of interface

Condition 8

Value	Meaning
0	Clear ATN
1	Set ATN asynchronously
2	Set ATN synchronously
Other	ERANGE error

Examples

For Pascal:

```
VAR
 err : INTEGER;

err := IOCONTROL (7,5,1); {Address the interface as talker.}
```

For C:

```
int     error;

error = IOCONTROL(7L,5,1); /*Address interface as talker.*/
```

Bus Activity

None.

Comments

Possible errors are NOERR, ESEL, ECTRL, ETIME, and ERANGE.

The added functionality for changing system controller status of the HP 82335 is included for completeness in the Command Library. We strongly recommend, however, that you do not use this command unless it is absolutely necessary. The recommended method of using the interface as a non-system controller is to use the MS-DOS command IOSYSCTL in your AUTOEXEC.BAT file.

For condition 8, you can set ATN either synchronously or asynchronously. Typically, you will set ATN asynchronously. If so, data may get lost if a data transfer is occurring that does not involve the HP 82335. For example, if a scope is talking to a printer and ATN is set asynchronously, some data may have been lost. If you want to avoid this situation, use status 2 to set ATN synchronously.

Refer to the Comments section of the IOPASSCTL command for important information about using Ctrl-C and Ctrl-Break.

IODMA

This command sets up DMA control. Using DMA may decrease the time required to transfer longer sequences of data using IOENTERAB, IOENTERB, IOENTERS, IOOUTPUTAB, IOOUTPUTB, and IOOUTPUTS.

Syntax

`IODMA` (*select_code*, *value*, *channel*)

select_code specifies the interface on which DMA is to be enabled or disabled.

value specifies one of the following:

Value	**Action Taken**
zero	Disables DMA. This is the default value.
positive value	Transfer size. Determines when a DMA read or write is executed. For example, if *value* = 100, then DMA will be used when 100 or more bytes are to be read or written.
negative value	Illegal. Will return an error.

channel indicates which channel to use for DMA. If the channel is other than 2 or 3, an error is returned.

Examples

For Pascal:

```
VAR
 err : INTEGER;

err := IODMA (7,1000,3); {Enable DMA for string transfers of
                          1000 characters or more.}
```

For C:

```
int     error;

error = IODMA(7L,1000,3); /*Enable DMA for input or output
                            of 1000 or more characters.*/
```

Bus Activity

None.

Comments

DMA channel 3 is the recommended channel. This is least likely to conflict with established usage.

If character matching is enabled at the time IOENTERAB, IOENTERB, or IOENTERS using DMA is attempted, the error EUNKNOWN will be returned for that command and no data will be transferred.

If byte swapping is specified in IOENTERAB, IOENTERB, IOOUTPUTAB, or IOOUTPUTB using DMA (*swapsize* is greater than 1), the error EUNKNOWN will be returned for that command and no data will be transferred.

Possible errors are NOERR, ESEL, and ERANGE.

IOENTER

This command reads a single real number. Reading continues until one of these events occurs:

- The EOI line is sensed true, if it is enabled.
- A linefeed is encountered after the number starts.

Numeric characters are the digits 0 through 9, "E", "e", "+", "−", and "." in the proper sequence for representing a number. Note that " " (space) is not a numeric character.

Syntax

IOENTER (*device_address*, *data_REF*)
IOENTER (*select_code*, *data_REF*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data_REF	variable into which the reading is placed.

Examples

For Pascal:

```
VAR
 reading : REAL;
     err : INTEGER;

err := IOENTER (722,reading); {Input a number from device
                               722 and place it in READING.}
```

For C:

```
float     reading;
int       error;

error = IOENTER(722L,&reading); /*Input a number from
                                  device 722.*/
```

Bus Activity

If a device address is specified:

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.

If a select code is specified:

- If the interface is not addressed to listen, an error results.
- If the interface is addressed to listen, ATN is cleared and the data is read from the interface.

Comments

If a select code is to be specified in the command, the interface must first be addressed to listen (with IOSEND, for example) or an error occurs.

The approximate range of valid values is 10^{-38} to 10^{38}. The IEEE 754 standard for floating point numbers makes provisions for values less than 10^{-38}, however the internal number conversion may not properly handle values less than 10^{-38} when entered via HP-IB or used in assignment or print statements.

Possible errors are NOERR, ETIME, ESEL, EADDR, ECTRL, and ENUM.

IOENTERA

This command enters numbers from a device or the interface and places them into a real array. Reading continues until one of these events occurs:

- The EOI line is sensed true, if it is enabled.
- A linefeed is encountered after the specified number of elements is received.

Numeric characters are the digits 0 through 9, "E", "e", "+", "–", and "." in the proper sequence for representing a number. Note that " " (space) is not a numeric character.

Syntax

`IOENTERA` (*device_address*, *readings_REF*, *elements_REF*)
`IOENTERA` (*select_code*, *readings_REF*, *elements_REF*)

device_address	specifies a device address.
select_code	specifies the interface select code.
readings_REF	array into which the readings are placed.
elements_REF	variable that specifies the maximum number of elements to be read. (An error occurs if the number is less than 0.) Upon return it indicates the number of elements actually received.

Examples

For Pascal:

```
TYPE
 REALARRAY = SUPER ARRAY[1..*] of REAL; {From IODECL.EX}
                             {For Turbo Pascal,
                              real50 = ARRAY[1..50] of REAL;}
VAR
 readings : REALARRAY(50); {For Turbo Pascal,
                              readings : real50;}
 elements : INTEGER;
      err : INTEGER;

elements := 50;
err := IOENTERA (723,readings,elements); {Read a maximum of
       50 values from device 723 and put them in READINGS.}
```

For C:

```
float     readings[50];
int       elements;
int       error;

elements = 50;
error = IOENTERA(723L,readings,&elements);
        /*Read a maximum of 50 values from device 723.*/
```

Bus Activity

If a device address is specified:

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.

If a select code is specified:

- If the interface is not addressed to listen, an error results.
- If the interface is addressed to listen, ATN is cleared and the data is read from the interface.

Comments

If the specified maximum number of elements to read is greater than the size of the *readings* array, the size of the array is used as the maximum number for Microsoft Pascal only—for other languages, input data can overrun the array and corrupt existing data or programs.

Nonnumeric characters that do not properly belong in a real number are considered value separators. Thus, the sequence "1,234,567" is entered as three numbers, not as "1234567".

The number of readings available is dependent upon the source device.

The approximate range of valid values is 10^{-38} to 10^{38}. The IEEE 754 standard for floating point numbers makes provisions for values less than 10^{-38}, however the internal number conversion may not properly handle values less than 10^{-38} when entered via HP-IB or used in assignment or print statements.

If a select code is to be specified in the command, the interface must first be addressed to listen (with IOSEND, for example) or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ENUM, ECTRL, and ERANGE.

IOENTERAB

This command enters arbitrary-block program data (numeric or string data with IEEE-488.2 coding) from a device or the interface. Reading continues until one of these events occurs:

- The maximum number of bytes specified is received.
- A linefeed is encountered with the EOI line sensed true, if the coding indicates indefinite length.
- The number of bytes indicated by the coding is received, if the coding indicates definite length.

Syntax

IOENTERAB (*device_address*, *data_REF*, *bytes_REF*, *swapsize*)
IOENTERAB (*select_code*, *data_REF*, *bytes_REF*, *swapsize*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data_REF	array into which the readings are placed.
bytes_REF	variable specifying the maximum number of bytes to be read (excluding the coding bytes). (An error occurs if the number is less than 0.) Upon return it indicates the number of bytes actually received (excluding the coding bytes).
swapsize	specifies how bytes are placed into memory. A value of 1 indicates that bytes are placed in order. Larger values indicate that bytes are reversed in memory in groups of this size. The value should correspond to the byte size of the *data* variable. (For example, a value of 4 specifies that each group of four bytes is swapped in memory.) Valid values are 1 through 8—other values return an error.

Examples

For Pascal:

```
TYPE
 BINDOUBLE = SUPER ARRAY[1..*] of REAL8;
 {Double-precision array (8 bytes/elem) from IODECL.EX}
                        {For Turbo Pascal,
                         double50 = ARRAY[1..50] of REAL8;}
VAR
      val : BINDOUBLE(50);     {For Turbo Pascal,
                                val : double50;}
 elements : INTEGER;
     swap : INTEGER;
      err : INTEGER;

swap := 8;
elements := 50 * swap;
err := IOENTERAB (723,val,elements,swap); {Read a maximum
       of 50 values from device 723 and put them in VAL.}
```

For C:

```
double     val[50]; /*Double-precision array (8 bytes/elem)*/
int        elements;
int        swap;
int        error;

swap = sizeof(double);
elements = 50 * swap;
error = IOENTERAB(723L,val,&elements,swap);
        /*Read a maximum of 50 values from device 723.*/
```

Bus Activity

If a device address is specified:

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.

If a select code is specified:

- If the interface is not addressed to listen, an error results.
- If the interface is addressed to listen, ATN is cleared and the data is read from the interface.

Comments

IEEE-488.2 coding is described under "Arbitrary-Block Data Coding" in chapter 1. The coding bytes are not placed into *data*—this also applies to the ending linefeed character for indefinite-length data. Leading characters are ignored until a "#" character is received.

If the specified maximum number of elements to read is greater than the size of the *data* array, input data can overrun the array and corrupt existing data or programs.

For Microsoft Pascal, you can use only one array type with IOENTERAB in a program. The IOPROC.EX file declares the type as a real array. If you want to use another type, edit IOPROC.EX to make the appropriate declaration—other types are included as comments in the file.

For string transfers, only the string elements receiving data are affected. The string descriptor and other string elements remain unchanged for Pascal—no null character is appended for C.

If DMA is active for the transfer, the *swapsize* parameter must be 1 and character matching must be disabled—otherwise, an EUNKNOWN error occurs.

The number of bytes available is dependent upon the source device.

If a select code is to be specified in the command, the interface must first be addressed to listen (with IOSEND, for example) or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ERANGE, ECTRL, and EUNKNOWN.

IOENTERB

This command enters binary data (numeric or string data with no coding or formatting) from a device or the interface. Reading continues until one of these events occurs:

- The maximum number of bytes specified is received.
- The EOI line is sensed true, if it is enabled.
- The termination character set by IOMATCH is received with EOI true. (Linefeed is the default character.)

Syntax

`IOENTERB` (*device_address*, *data_REF*, *bytes_REF*, *swapsize*)
`IOENTERB` (*select_code*, *data_REF*, *bytes_REF*, *swapsize*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data_REF	array into which the readings are placed.
bytes_REF	specifies the maximum number of bytes to be read. (An error occurs if the number is less than 0.) Upon return it indicates the number of bytes actually received.
swapsize	specifies how bytes are placed into memory. A value of 1 indicates that bytes are placed in order. Larger values indicate that bytes are reversed in memory in groups of this size. The value should correspond to the byte size of the *data* variable. (For example, a value of 4 specifies that each group of four bytes is swapped in memory.) Valid values are 1 through 8—other values return an error.

Examples

For Pascal:

```
TYPE
 BINDOUBLE = SUPER ARRAY[1..*] of REAL8;
 {Double-precision array (8 bytes/elem) from IODECL.EX}
                       {For Turbo Pascal,
                        double50 = ARRAY[1..50] of REAL8;}
VAR
      val : BINDOUBLE(50);     {For Turbo Pascal,
                                val : double50;}
 elements : INTEGER;
     swap : INTEGER;
      err : INTEGER;

swap := 8;
elements := 50 * swap;
err := IOENTERB (723,val,elements,swap); {Read a maximum
       of 50 values from device 723 and put them in VAL.}
```

For C:

```
double     val[50]; /*Double-precision array (8 bytes/elem)*/
int        elements;
int        swap;
int        error;

swap = sizeof(double);
elements = 50 * swap;
error = IOENTERB(723L,val,&elements,swap);
        /*Read a maximum of 50 values from device 723.*/
```

Bus Activity

If a device address is specified:

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.

If a select code is specified:

- If the interface is not addressed to listen, an error results.
- If the interface is addressed to listen, ATN is cleared and the data is read from the interface.

Comments

If the specified maximum number of elements to read is greater than the size of the *data* array, input data can overrun the array and corrupt existing data or programs.

For Microsoft Pascal, you can use only one array type with IOENTERB in a program. The IOPROC.EX file declares the type as a real array. If you want to use another type, edit IOPROC.EX to make the appropriate declaration—other types are included as comments in the file.

All data received is stored in memory—except a final "match" character with EOI true if matching is enabled.

For string transfers, only the string elements receiving data are affected. The string descriptor and other string elements remain unchanged for Pascal—no null character is appended for C.

If DMA is active for the transfer, the *swapsize* parameter must be 1 and character matching must be disabled—otherwise, an EUNKNOWN error occurs.

The number of bytes available is dependent upon the source device.

If a select code is to be specified in the command, the interface must first be addressed to listen (with IOSEND, for example) or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ERANGE, ECTRL, and EUNKNOWN.

IOENTERF

This command reads from a device and places all received data into a file. Reading continues until one of these events occurs:

- The EOI line is sensed true, if it is enabled.
- The termination character set by IOMATCH is received (linefeed is the default). Note: If you are transferring binary files, you should turn off character match using IOMATCH to make sure the transfer does not end prematurely.
- The maximum number of bytes specified is received.
- A file error occurs, usually meaning the disk is full.

Computer Museum

Syntax

IOENTERF (*device_address*,*file_name*,*length*,*append_flag*)
IOENTERF (*select_code*,*file_name*,*length*,*append_flag*)

device_address	specifies a device address.
select_code	specifies the interface select code.
file_name	file into which the data is written.
length	specifies the maximum number of elements to be read. (An error occurs if the number is less than 0.) The actual number of bytes read is returned here.
append_flag	specifies whether to append to the file or to overwrite it. Zero overwrites; non-zero appends.

Examples

For Pascal:

```
error  : INTEGER ;
length : LONGINT ;   (* for Turbo Pascal *)
length : INTEGER4 ;  (* for Microsoft Pascal *)

length := 10;
error := IOENTERF(723, 'ENTERF.DAT',length, 0)
if error <> NOERR then writeln('an error occurred...');
```

For C:

```
int   error;
long  length;

length = 10;
error = IOENTERF(723L, "ENTERF.DAT", &length, 0)
if (error != NOERR) printf ("an error occurred...\n");
```

Bus Activity

If a device address is specified:

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.

If a select code is specified:

- If the interface is not addressed to listen, an error results.
- If the interface is addressed to listen, ATN is cleared and the data is read from the interface.

Comments

Possible errors are NOERR, ETIME, ESEL, EADDR, ERANGE, ECTRL, and EFILE.

If the file does not exist, and a valid filename is given, IOENTERF will create the file regardless of the append flag.

We recommend turning off character matching using the IOMATCH command, especially if you are transferring a binary file.

Note

This command does **not** transfer files to an HP-IB disk drive, but rather transfers bytes from the HP-IB bus to a built-in disk drive on your computer.

IOENTERS

This command enters a character string from a device or the interface. Reading continues until one of these events occurs:

- The EOI line is sensed true, if it is enabled.
- The termination character set by IOMATCH is received (linefeed is the default).
- The maximum number of characters specified is received.

Syntax

IOENTERS (*device_address*, *data_REF*, *length_REF*)
IOENTERS (*select_code*, *data_REF*, *length_REF*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data_REF	variable into which the string read is placed.
length_REF	variable specifying the maximum number of elements to be read. (An error occurs if the number is less than 0.) On return it indicates the number of elements actually received.

Examples

For Pascal:

```
VAR
 info   : STRING(10);
 length : INTEGER;
 err    : INTEGER;

length := 10;
err := IOENTERS (723,info,length);
       {Read a string of 10 characters maximum from device 723.}
```

For C:

```
int     error;
int     length;
char    info[11];  /*10 characters plus null*/

length = 10;
error = IOENTERS(723L,info,&length); /*Read a string of
        10 characters maximum from device 723.*/
```

Bus Activity

If a device address is specified:

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.

If a select code is specified:

- If the interface is not addressed to listen, an error results.
- If the interface is addressed to listen, ATN is cleared and the data is read from the interface.

Comments

If the specified maximum number of elements to read is greater than the dimensioned length of the *data* string:

- For Pascal, the dimensioned length is used instead of the maximum number.
- For C, input data can overrun the string and corrupt existing data or programs.

If a select code is to be specified in the command, the interface must first be addressed to listen (with IOSEND or a previous IOENTER, for example) or an error occurs.

The termination character is entered as part of the string. For C, a null character is appended to the string.

If DMA is active for the transfer, character matching must be disabled—otherwise, an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ENUM, ERANGE, ECTRL, and EUNKNOWN.

IOEOI

This command enables or disables the End Or Identify (EOI) mode of the interface. It is used to:

- Enable or disable a write operation to set the EOI line on the last byte of the write.
- Enable or disable a read operation to terminate upon sensing the EOI line true.

The default is EOI enabled.

Syntax

`IOEOI` (*select_code*, *state*)

select_code	specifies the interface select code.
state	enables EOI if nonzero and disables EOI if zero.

Examples

For Pascal:

```
VAR
 state : INTEGER;
   err : INTEGER;

state := 0;
  err := IOEOI(7,state); {Disable EOI.}
```

For C:

```
int     error;

error = IOEOI(7L,0); /*Disable EOI.*/
```

Bus Activity

None.

Comments

When reading with EOI enabled, receipt of a byte with EOI set causes the read operation to terminate, regardless of whether you are reading a string, a real number, or an array of real numbers. (The EOI state is ignored by IOENTERAB.)

When writing, EOI is set on the last byte of the End Of Line sequence if EOI is enabled. Note that if the EOL sequence is of 0 length, EOI is set on the last data byte sent. (The EOI line is *not* set on the last byte for IOOUTPUTAB.)

When sending a real number array with IOOUTPUTA, the EOL sequence (and subsequent EOI) is appended after the last element in the array, not after each element.

Note that IOSEND does not set EOI because this line has a different meaning in Command mode.

Possible errors are NOERR and ESEL.

IOEOL

This command defines the End of Line (EOL) string that is to be sent following every IOOUTPUT, IOOUTPUTA, IOOUTPUTB, and IOOUTPUTS command.

The default is carriage return and linefeed.

Syntax

`IOEOL` (*select_code*, *endline_REF*, *length*)

select_code	specifies the interface select code.
endline_REF	specifies the EOL string that is to be sent following a data transmission. A maximum of eight characters can be specified.
length	specifies the length of the termination string. If zero is specified, no characters are appended to a data transmission. If the length is less than 0 or more than 8, an error occurs.

Examples

For Pascal:

```
VAR
  length : INTEGER;
 endline : STRING(2);
     err : INTEGER;

    length := 2;
endline[1] := CHR(13);
endline[2] := CHR(10);
       err := IOEOL(7,endline,length); {EOL = CR/LF.}
```

For C:

```
int     length;
char    endline[2];
int     error;

length = 2;
endline[0] = 13;
endline[1] = 10;
error = IOEOL(7,endline,length); /*EOL  =  CR/LF.*/
```

Bus Activity

None.

Comments

With IOOUTPUTA and IOOUTPUTB, the EOL sequence is appended after all data has been sent, not following each element.

Possible errors are NOERR, ESEL, and ERANGE.

IOFASTOUT

This command enables or disables high-speed bus timing for output transfers only.

The default is high-speed output disabled (standard speed).

Syntax

`IOFASTOUT` (*select_code*, *state*)

select_code	specifies the interface select code.
state	enables high-speed output if nonzero and disables high-speed output if zero.

Examples

For Pascal:

```
VAR
 state : INTEGER;
   err : INTEGER;

state := 0;
  err := IOFASTOUT(7,state); {Disable high-speed output.}
```

For C:

```
int     error;

error = IOFASTOUT(7L,0); /*Disable high-speed output.*/
```

Bus Activity

None.

Comments

For proper operation, high-speed output requires the HP-IB system to meet all of these requirements:

- All HP-IB devices must have tri-state drivers, not open-collector drivers. (The HP-IB interface meets this requirement.)
- All HP-IB devices must be turned on.
- HP-IB cable length should be as short as possible, but not longer than 15 meters (50 feet). At least one HP-IB device should be connected for each meter (3 feet) of cable, with a maximum of 15 devices. (The HP-IB interface counts as one device.)
- Each HP-IB device must have a capacitance of less than 50 pF on each HP-IB line except REN and IFC. (The HP-IB interface meets this requirement.)

High-speed output applies only during output transfers (including DMA output transfers)—but not between transfers and not during input transfers. The speed of an input transfer depends upon the talker device.

High-speed output decreases the data-settling time from 2.5 microseconds to 840 nanoseconds.

Possible errors are NOERR and ESEL.

IOGETTERM

This command determines the reason the last read terminated.

Syntax

`IOGETTERM` (*select_code*, *reason_REF*)

select_code	specifies the interface select code.
reason_REF	variable to receive the sum of the values for the reasons the last read terminated. The possible reasons for termination are

Value	Description
0	The read was terminated for some reason not covered by any of the other reasons.
1	The expected number of elements was received.
2	The termination character set by IOMATCH was encountered.
4	The EOI line was sensed true.

Examples

For Pascal:

```
VAR
 reason : INTEGER;
    err : INTEGER;

err := IOGETTERM(7,reason);
IF ((reason and 4) = 4) then
  WRITELN('EOI ENCOUNTERED');
```

For C:

```
int     reason;
int     error;

error = IOGETTERM(7L,&reason);
if((reason & 4) == 4)
    printf("EOI ENCOUNTERED\n");
```

Bus Activity

None.

Comments

Upon return, the reason integer contains the sum of the values for the reasons for termination. For example, if the last read terminated when the termination character was encountered and EOI was set, the value of reason would be $2 + 4 = 6$.

Possible errors are NOERR and ESEL.

IOLLOCKOUT

This command sends a Local Lockout (LLO) to disable a device front panel. It is received by all devices on the interface, whether or not they are addressed to listen.

Syntax

`IOLLOCKOUT` (*select_code*)

select_code	specifies the interface select code.

Examples

For Pascal:

```
VAR
 err : INTEGER;

 err := IOLLOCKOUT (7);
```

For C:

```
int     error;

error = IOLLOCKOUT(7L);
```

Bus Activity

- ATN is sent.
- LLO is sent.

Comments

If a device is in Local mode when LLO is received, LLO does not take effect until the device is addressed to listen.

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

IOLOCAL

This command executes a Go To Local (GTL) or clears the REN line to enable a device front panel.

Syntax

IOLOCAL (*device_address*)
IOLOCAL (*select_code*)

device_address	specifies a device address.
select_code	specifies the interface select code.

Examples

For Pascal:

```
VAR
 err : INTEGER;

err := IOLOCAL (722); {Place device 722 in Local mode.}
```

For C:

```
int     error;

error = IOLOCAL(722L); /*Place device 722 in local mode.*/
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- GTL is sent.

If a select code is specified:

- REN is cleared.
- ATN is cleared.

Comments

If a device address is specified, the device is temporarily placed in Local mode—it will return to Remote mode if it is later addressed to listen. If Local Lockout is in effect, the device will return to the Lockout state if it is later addressed to listen.

If an interface select code is specified, all instruments on the bus are placed in Local mode and any Local Lockout is cancelled.

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

IOMATCH

This command defines the character used by IOENTERB and IOENTERS for termination.

The default character is linefeed.

Syntax

`IOMATCH` (*select_code*, *character*, *flag*)

select_code	specifies the interface select code.
character	specifies the character used by IOENTERB and IOENTERS for termination checking.
flag	indicates whether character matching should be enabled or disabled. Zero disables matching, and any nonzero value enables it.

Examples

For Pascal:

```
VAR
 match : CHAR;
 flag  : INTEGER;
   err : INTEGER;

match := CHR(10); {Terminate on linefeed.}
 flag := 1;
  err := IOMATCH (7,match,flag);
```

For C:

```
char    match;
int     flag;
int     error;

match = 10; /*Terminate on linefeed.*/
flag = 1;
error = IOMATCH(7L,match,flag);
```

Bus Activity

None.

Comments

Only a single match character may be specified in this command.

For IOENTERS, the match character becomes part of the entered string. For IOENTERB, the match character must be received with EOI true, and the character does *not* become part of the data.

IOMATCH does not apply to IOENTER, IOENTERA, or IOENTERAB.

Possible errors are NOERR and ESEL.

IOOUTPUT

This command outputs a real number to a device or to the interface. After the number is sent, the EOL string is sent and the EOI line is set (if enabled).

Syntax

IOOUTPUT (*device_address*, *data*)
IOOUTPUT (*select_code*, *data*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data	specifies the number to be output.

Examples

For Pascal:

```
VAR
 data : REAL;
  err : INTEGER;

data := 12.3;
 err := IOOUTPUT (722,data); {Output ' 12.3' to dev 722.}
```

For C:

```
double    data;
int       error;

data = 12.3;
error = IOOUTPUT(722L,data); /*Output ' 12.3' to dev 722.*/
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is output.
- EOL is output.

If a select code is specified:

- If the interface is not addressed to talk, an error results.
- If the interface is addressed to talk, ATN is cleared and the data is sent followed by an EOL.

Comments

Numbers with absolute values between 10^{-5} and 10^{6} are rounded to seven significant digits and output in floating point notation. If the number rounds to an integer value, the decimal point is not sent. Numbers outside this range are rounded to seven significant digits and output in scientific notation.

If the number is positive, a leading space is output for the sign; if it's negative, a leading "–" is output.

If a select code is to be specified, the interface must first be addressed to talk (with IOSEND, for example), or an error occurs.

Possible errors are NOERR, ETIME, ESEL, ECTRL, and EADDR.

IOOUTPUTA

This command outputs an array of real numbers to a specified device or to the bus. Values output are separated by commas. After the last number is sent, the EOL string is sent and the EOI line is set (if enabled).

Syntax

`IOOUTPUTA` (*device_address*, *data_REF*, *elements*)
`IOOUTPUTA` (*select_code*, *data_REF*, *elements*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data_REF	array containing the real numbers to be transmitted.
elements	specifies the number of elements in the array to be transmitted. (An error occurs if the number is less than 0.)

Examples

For Pascal:

```
TYPE
 REALARRAY = SUPER ARRAY[1..*] OF REAL; {From IODECL.EX.}
                            {For Turbo Pascal,
                             real10 = ARRAY[1..10] of REAL;}
VAR
         info : REALARRAY(10);  {For Turbo Pascal,
                                  info : real10;}
 num_elements : INTEGER;
          err : INTEGER;

num_elements := 10;
err := IOOUTPUTA (722,info,num_elements);
       {Output the array INFO to device 722.}
```

For C:

```
float    info[10];
int      num_elements;
int      error;

num_elements = 10;
error = IOOUTPUTA(722L,info,num_elements);
        /*Output elements of 'info' to device 722.*/
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is output.
- EOL is output.

Computer Museum

If a select code is specified:

- If the interface is not addressed to talk, an error results.
- If the interface is addressed to talk, ATN is cleared and the data is sent followed by an EOL.

Comments

If the specified maximum number of elements to output is greater than the size of the *data* array, the array size is used as the maximum number for Microsoft Pascal only—for other languages, the output transfer can go beyond the array and send meaningless data.

Numbers with absolute values between 10^{-5} and 10^{6} are rounded to seven significant digits and output in floating point notation. If the number rounds to an integer value, the decimal point is not sent. Numbers outside this range are rounded to seven significant digits and output in scientific notation.

If the number is positive, a leading space is output for the sign; if it's negative, a leading "–" is output.

If a select code is to be used as a parameter, the interface must first be addressed to talk (with IOSEND, for example), or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ECTRL, and ERANGE.

IOOUTPUTAB

This command outputs arbitrary-block response data (numeric or string data with IEEE-488.2 coding) to a specified device or to the bus. After the last data byte is sent, nothing additional occurs.

Syntax

IOOUTPUTAB (*device_address*, *data_REF*, *bytes*, *swapsize*)
IOOUTPUTAB (*select_code*, *data_REF*, *bytes*, *swapsize*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data_REF	array containing the data to be transmitted.
bytes	specifies the number of bytes to output (excluding the coding bytes). This value should be no more than the number of elements in the array times the number of bytes per element. (An error occurs if the number is less than 0.)
swapsize	specifies how bytes are read from memory. A value of 1 indicates that bytes are read in order. Larger values indicate that bytes are reversed as read from memory in groups of this size. The value should correspond to the byte size of the *data* variable. (For example, a value of 4 specifies that each group of four bytes is swapped when output.) Valid values are 1 through 8—other values return an error.

Examples

For Pascal:

```
TYPE
 BINDOUBLE = SUPER ARRAY[1..*] of REAL8;
 {Double-precision array (8 bytes/elem) from IODECL.EX}
                       {For Turbo Pascal,
                        double10 = ARRAY[1..10] of REAL8;}
VAR
       val : BINDOUBLE(10);     {For Turbo Pascal,
                                 val : double10;}
 num_bytes : INTEGER;
      swap : INTEGER;
       err : INTEGER;

swap := 8;
num_bytes := 10 * swap;
err := IOOUTPUTAB (722,info,num_elements,swap);
       {Output the array INFO to device 722.}
```

For C:

```
double   info[10]; /*Double-precision array (8 bytes/elem)*/
int      num_bytes;
int      swap;
int      error;

swap = sizeof(double);
num_bytes = 10 * swap;
error = IOOUTPUTAB(722L,info,num_bytes,swap);
        /*Output elements of 'info' to device 722.*/
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is output.

If a select code is specified:

- If the interface is not addressed to talk, an error results.
- If the interface is addressed to talk, ATN is cleared and the data is sent.

Comments

IEEE-488.2 coding is described under "Arbitrary-Block Data Coding" in chapter 1. The coding bytes are automatically computed and inserted in front of the data.

If the specified maximum number of elements to output is greater than the size of the *data* array, the output transfer can go beyond the array and send meaningless data.

For Microsoft Pascal, you can use only one array type with IOOUTPUTAB in a program. The IOPROC.EX file declares the type as a real array. If you want to use another type, edit IOPROC.EX to make the appropriate declaration—other types are included as comments in the file.

If DMA is active for the transfer, the *swapsize* parameter must be 1—otherwise, an EUNKNOWN error occurs.

If a select code is to be specified in the command, the interface must first be addressed to talk (with IOSEND, for example) or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ERANGE, ECTRL, and EUNKNOWN.

IOOUTPUTB

This command outputs binary data (numeric or string data with no coding or formatting) to a specified device or to the bus. After the last number is sent, the EOL string is sent and the EOI line is set (if enabled).

Syntax

IOOUTPUTB (*device_address*, *data_REF*, *bytes*, *swapsize*)
IOOUTPUTB (*select_code*, *data_REF*, *bytes*, *swapsize*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data_REF	array containing the data to be transmitted.
bytes	specifies the number of bytes to output. This value should be no more than the number of elements in the array times the number of bytes per element. (An error occurs if the number is less than 0.)
swapsize	specifies how bytes are read from memory. A value of 1 indicates that bytes are read in order. Larger values indicate that bytes are reversed as read from memory in groups of this size. The value should correspond to the byte size of the *data* variable. (For example, a value of 4 specifies that each group of four bytes is swapped when output.) Valid values are 1 through 8—other values return an error.

Examples

For Pascal:

```
TYPE
 BINDOUBLE = SUPER ARRAY[1..*] of REAL8;
 {Double-precision array (8 bytes/elem) from IODECL.EX}
                        {For Turbo Pascal,
                         double10 = ARRAY[1..10] of REAL8;}
VAR
       val : BINDOUBLE(10);      {For Turbo Pascal,
                                  val : double10;}
 num_bytes : INTEGER;
      swap : INTEGER;
       err : INTEGER;

swap := 8;
num_bytes := 10 * swap;
err := IOOUTPUTB (722,info,num_elements,swap);
       {Output the array INFO to device 722.}
```

For C:

```
double   info[10]; /*Double-precision array (8 bytes/elem)*/
int      num_bytes;
int      swap;
int      error;

swap = sizeof(double);
num_bytes = 10 * swap;
error = IOOUTPUTB(722L,info,num_bytes,swap);
        /*Output elements of 'info' to device 722.*/
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is output.
- EOL is output.

If a select code is specified:

- If the interface is not addressed to talk, an error results.
- If the interface is addressed to talk, ATN is cleared and the data is sent followed by an EOL.

Comments

If the specified maximum number of elements to output is greater than the size of the *data* array, the output transfer can go beyond the array and send meaningless data.

For Microsoft Pascal, you can use only one array type with IOOUTPUTB in a program. The IOPROC.EX file declares the type as a real array. If you want to use another type, edit IOPROC.EX to make the appropriate declaration—other types are included as comments in the file.

If DMA is active for the transfer, the *swapsize* parameter must be 1—otherwise, an EUNKNOWN error occurs.

If a select code is to be specified in the command, the interface must first be addressed to talk (with IOSEND, for example), or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ERANGE, ECTRL, and EUNKNOWN.

IOOUTPUTF

This command outputs the contents of a file to a specified device or interface. After the file is sent, the EOL string is sent and the EOI line is set (if enabled).

Syntax

IOOUTPUTF (*device_address*,*file_name*,*length*)
IOOUTPUTF (*select_code*,*file_name*,*length*)

device_address	specifies a device address.
select_code	specifies the interface select code.
file_name	specifies the name of the file to output.
length	specifies the maximum number of elements to be written. (An error occurs if the number is less than 0.)

Examples

For Pascal:

```
error   : INTEGER ;
length  : LONGINT ;  (* for Turbo Pascal *)
length  : INTEGER4 ; (* for Microsoft Pascal *)

length := 10 ;
error := IOOUTPUTF(723, 'OUTPUT.DAT', length)
if error <> NOERR then writeln('an error occurred...');
```

For C:

```
int  error ;
long length ;

length = 10 ;
error = IOOUTPUTF(723L, "OUTPUT.DAT", &length)
if (error != NOERR) printf ("an error occurred...\n");
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is entered.
- EOL is output.

If a select code is specified:

- If the interface is not addressed to talk, an error results.
- If the interface is addressed to talk, ATN is cleared and the data is sent followed by the EOL string.

Comments

Possible errors are NOERR, ETIME, ESEL, EADDR, ERANGE, ECTRL, and EFILE.

If you are transferring a binary file, we recommend that you turn off the EOL string using the IOEOL command. If you do not, the current EOL string will be appended to the file.

Note

This command does **not** transfer files from an HP-IB disk drive, but rather transfers bytes from a built-in disk drive on your computer to the HP-IB bus.

IOOUTPUTS

This command outputs a string to a specified device or to the interface. After the string is sent, the EOL string is sent and the EOI line is set (if enabled).

Syntax

IOOUTPUTS (*device_address*, *data_REF*, *length*)
IOOUTPUTS (*select_code*, *data_REF*, *length*)

device_address	specifies a device address.
select_code	specifies the interface select code.
data_REF	array specifying the string to be sent.
length	specifies the length of the output string. (An error occurs if the number is less than 0.)

Examples

For Pascal:

```
VAR
   info : STRING(4);
 length : INTEGER;
    err : INTEGER;

info := '1ST1';
length := 4;
err := IOOUTPUTS (723,info,length);
       {Send the programming code '1ST1' to device 723.}
```

For C:

```
char    *info
int     length;
int     error;

info = "1ST1";
length = 4;
error = IOOUTPUTS(723L,info,length);
        /*Send the programming code '1ST1' to device 723.*/
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- ATN is cleared.
- Data is output.
- EOL is output.

If a select code is specified:

- If the interface is not addressed to talk, an error results.
- If the interface is addressed to talk, ATN is cleared and the data is sent followed by an EOL.

Comments

If the specified maximum number of elements to output is greater than the current length of the *data* string:

- For Pascal, the current length is used instead of the maximum number.
- For C, the output transfer can go beyond the string and send meaningless data.

If a select code is to be used in the command, the interface must first be addressed to talk (with IOSEND, for example), or an error occurs.

Possible errors are NOERR, ETIME, ESEL, EADDR, ECTRL, and ERANGE.

IOPASSCTL

This command passes active control from the HP 82335 HP-IB card to a device on the bus. The device must be capable of taking control.

Syntax

`IOPASSCTL` (*device_address*)

device_address specifies a device address.

Examples

For Pascal:

```
error  : INTEGER ;

error  := IOPASSCTL(723)
if error <> NOERR then writeln('an error occurred...');
```

For C:

```
int  error ;

signal (SIGINT, ctrlc_handler) ;  /* trap CTRL-C */

error = IOPASSCTL(723L)
if (error !=NOERR) printf ("an error occurred...\n");

void ctrlc_handler ()  /* used for CTRL-C interrupts */
{
     exit (1) ;  /* exit with error code 1 */
}
```

Bus Activity

If a device address is specified:

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- TCT is sent.
- ATN is cleared.

Comments

If your program does not seem to work properly after passing control, make sure that you do not have an interrupt (IRQ) conflict with another device. You can find out what IRQ your HP-IB board is using by running the INSTALL utility.

The Command Library defaults to address 30. If desired, you can change this using the IOCONTROL command.

The IOPASSCTL command passes active control only. This command will not change the state of the system controller status of the HP 82335.

Any type of *shell* command will cause the Command Library to stop working if it is currently non-controller, including the SYSTEM function in C languages and the EXEC function in Pascal languages.

The Command Library needs to do some cleaning up after running as non-active or non-system controller. It will do this automatically when you take control back, or when your program exits normally. It will not, however, clean up after itself if Ctrl-C or Ctrl-Break is used to exit the program. We recommend that you use the capabilities of the language you are using to trap these keys and call a routine which exits normally, possibly with a non-zero exit code. In C, use *signal* and *exit* if Ctrl-C is pressed. In Pascal, use *CheckBreak* and *halt*. If you do not, your computer will be left in an unstable state and could lock up.

Possible errors are NOERR, ETIME, ESEL, and ECTRL.

IOPPOLL

This command performs a parallel poll of the interface. It sets a variable to a value (0 to 255) representing the response of those instruments on the interface that respond to a parallel poll.

Syntax

`IOPPOLL` (*select_code*, *response_REF*)

select_code	specifies the interface select code.
response_REF	variable into which the parallel poll response byte is to be placed. The allowable range is 0 to 255. The eight bits of the response byte correspond to the eight HP-IB data lines (DIO1 through DIO8). Thus, a value of 32 would indicate that some device has responded to the parallel poll with a "1" on DIO6.

Examples

For Pascal:

```
VAR
 response : INTEGER;
      err : INTEGER;

err := IOPPOLL (7,response);
```

For C:

```
int     response;
int     error;

error = IOPPOLL(7L,&response);
```

Bus Activity

- ATN and EOI are asserted for 25 microseconds.
- The poll byte is read.
- EOI is cleared.
- ATN is restored to its previous state.

Comments

During a parallel poll, each enabled device may put a "1" on an assigned HP-IB data line according to its service request status—otherwise, the line is a "0". There are eight data lines (though more than one device may be assigned to one line). Using a parallel poll, several devices can indicate their service request status simultaneously. The *response* variable contains the state of the eight data lines: DIO1 (in bit 0) through DIO8 (in bit 7).

If the *response* variable contains a "1" in any bit, a device assigned to the corresponding HP-IB line has the service request status the device was set up to report. (See IOPPOLLC.) For example, a device may be set up to report on line DIO4 when it requests service. If an IOPPOLL command shows a "1" in bit 3 of *response*, your program knows the device needs service (assuming no other device is assigned to that line).

Not all devices are capable of responding to a parallel poll. Consult your particular device manuals for specifics.

Possible errors are NOERR, ECTRL, and ESEL.

IOPPOLLC

This command performs a Parallel Poll Configure. In preparation for a parallel poll command, it tells an instrument how to respond affirmatively to the parallel poll, and on which data line to respond.

In general, it sets a parallel poll response byte to reflect the response of a desired arrangement of instruments. Typically, you could define the bits to reflect the responses of particular instruments, or the result of a logical OR operation on several instrument responses.

Refer to IOPPOLL for more information.

Syntax

`IOPPOLLC` (*device_address*, *configuration*)

`IOPPOLLC` (*select_code*, *configuration*)

device_address	specifies the bus address of the device to be configured.
select_code	specifies the interface select code.
configuration	sent to the specified device indicating how it's to respond to a parallel poll. (See "Comments" below.)

Examples

For Pascal:

```
VAR
 configuration : INTEGER;
           err : INTEGER;

configuration := 10; {Respond with a '1' on line DIO3.}

err := IOPPOLLC (723,configuration);
```

For C:

```
int     error;
int     configuration;

configuration = 10; /*Respond with a '1' on line DIO3.*/
error = IOPPOLLC(723L,configuration);
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- PPC is sent.
- PPE is sent.

If a select code is specified:

- PPC is sent.
- PPE is sent.

Comments

The *configuration* parameter defines both the HP-IB line on which to respond and the service request status to indicate. It represents an eight-bit byte described below.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	Response (0 or 1)	DIO Line (DIO1 to DIO8)		

Bit 3 specifies the meaning of an affirmative response. Bits 2 through 0 specify the data line (DIO8 through DIO1). The valid range for *configuration* is 0 to 15—other values cause an error.

Parallel Poll Configuration	Bits	Value
Affirmative response for service request	00001xxx	8
Affirmative response for no service request	00000xxx	0
Respond on line DIO8	0000x111	7
Respond on line DIO7	0000x110	6
Respond on line DIO6	0000x101	5
Respond on line DIO5	0000x100	4
Respond on line DIO4	0000x011	3
Respond on line DIO3	0000x010	2
Respond on line DIO2	0000x001	1
Respond on line DIO1	0000x000	0

For example, to set up a device to indicate an affirmative response ("1") on line DIO5 if it needs service, the configuration value would be 8 + 4 = 12. Alternatively, for the device to indicate an affirmative response ("1") on line DIO5 when it *doesn't* need service, the configuration value would be 0 + 4 = 4.

Not all devices can be configured to respond to a parallel poll. Consult your particular device manuals for specifics.

Possible errors are NOERR, ETIME, ESEL, ECTRL, and ERANGE.

IOPPOLLU

This command performs a Parallel Poll Unconfigure (PPU). It directs an instrument to not respond to a parallel poll. It can be addressed to the interface or a specific device. Refer to IOPPOLLC for more information.

Syntax

IOPPOLLU (*device_address*)
IOPPOLLU (*select_code*)

device_address	specifies a device address.
select_code	specifies the interface select code.

Examples

For Pascal:

```
VAR
 err : INTEGER;

err := IOPPOLLU (722);
```

For C:

```
int     error;

error = IOPPOLLU(722L);
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- PPC is sent.
- PPD is sent.

If a select code is specified:

- ATN is sent.
- PPU is sent.

Comments

Some devices cannot be unconfigured from the bus. Consult your particular device manuals for specifics.

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

IOREMOTE

This command places a device in Remote mode to disable the device front panel. It can be addressed to the interface or to a specific device.

Syntax

IOREMOTE (*device_address*)
IOREMOTE (*select_code*)

device_address	specifies a device address.
select_code	specifies the interface select code.

Examples

For Pascal:

```
VAR
 err : INTEGER;

err := IOREMOTE (723); {Places device 723 in Remote.}

err := IOREMOTE (7);   {Set the interface REN line.}
```

For C:

```
int     error;

error = IOREMOTE(723L); /*Place device 723 in Remote.*/
    .
    .
error = IOREMOTE(7L); /*Set the interface REN line.*/
```

Bus Activity

If a device address is specified:

- REN is set.
- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.

If a select code is specified, then REN is set.

Comments

If a select code is specified, a device will not switch into Remote mode until it is addressed to listen.

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

IOREQUEST

This command sets up a serial poll status byte for the HP 82335 and optionally asserts the Service Request (SRQ) line.

Syntax

IOREQUEST (*select_code*, *status*)

select_code	specifies the interface select code.
status	specifies the serial poll status byte. If bit 6 in the status byte is set, the SRQ line will be asserted. If bit 6 is clear, SRQ will not be asserted.

Examples

For Pascal:

```
error  : INTEGER ;

error := IOTAKECTL(7, $042)   (* for Turbo Pascal *)
error := IOTAKECTL(7, #042)   (* for Microsoft Pascal *)
if error <> NOERR then writeln('an error occurred...');
```

For C:

```
int  error ;

error = IOTAKECTL(7L, 0x42)
if (error !=NOERR) printf ("an error occurred...\n");
```

Bus Activity

If bit 6 is set in the status parameter: SRQ is asserted.

If bit 6 is clear in the status parameter: no bus activity.

Comments

The HP 82335 interface must not be active controller, or else an ECTRL error will result.

Possible errors are NOERR, ECTRL, and ESEL.

IORESET

This command sets the interface to its start-up state, in which it is not listening and not talking.

In addition, it sets the following HP-IB parameters as indicated:

- The interface timeout is set to 0 seconds (the timeout is disabled).
- The interface EOI mode is enabled.
- High-speed data output is disabled.
- CR/LF is set as the EOL default.
- LF is set as the IOMATCH default.
- If the interface was system controller, then it will also become active controller.

Syntax

`IORESET` (*select_code*)

select_code	specifies the interface select code.

Examples

For Pascal:

```
VAR
 err : INTEGER;

err := IORESET (7);
```

For C:

```
int     error;

error = IORESET(7L);
```

Bus Activity

If the interface is system controller:

- IFC is pulsed (at least 100 microseconds).
- REN is cleared (at least 100 microseconds).
- ATN is cleared.

If the interface is non-system controller:

- No bus activity.

Comments

This command returns all devices on the interface to local (front panel) control.

Possible errors are NOERR and ESEL.

IOSEND

This command sends any sequence of user-specified HP-IB commands to the interface. For example, to send an output command to several instruments simultaneously, you can establish their talk/listen status with the IOSEND command, then issue the output command specifying a select code rather than a device address.

Syntax

`IOSEND` (*select_code*, *commands_REF*, *length*)

select_code	specifies the interface select code.
commands_REF	specifies a string of characters, each of which is treated as an interface command.
length	specifies the number of characters in the command string. (An error occurs if the number is less than 0.)

Examples

For Pascal:

```
VAR
 commands : STRING[4];
   length : INTEGER;
      err : INTEGER;

commands := '?)/4'; {Specifies unlisten, then
            listen addresses 9, 15, and 20.}
  length := 4;
     err := IOSEND (7,commands,length);
     err := IOTRIGGER (7);
            {Triggers devices at addresses 9, 15, and 20.}
```

For C:

```
char     *commands;
int      length;
int      error;

commands = "?)/4"; /*Specifies unlisten, then
            listen addresses 9, 15, and 20.*/
length = 4;
error = IOSEND(7L,commands,length);
   .
   .
error = IOTRIGGER(7L);
        /*Triggers devices at addresses 9, 15, and 20.*/
```

Bus Activity

- ATN is set.
- Command bytes are sent.

Comments

See appendix B for a list of HP-IB commands and the corresponding data characters.

All bytes are sent with ATN set. The EOL sequence is not appended, nor is EOI set.

Possible errors are NOERR, ETIME, ESEL, ECTRL, and ERANGE.

IOSPOLL

This command performs a serial poll of a specified device. It sets a variable representing the device's response byte.

Syntax

IOSPOLL (*device_address*, *response_REF*)

device_address	specifies the bus address of the device to be polled.
response_REF	variable into which the response byte is placed.

Examples

For Pascal:

```
VAR
 response : INTEGER;
      err : INTEGER;

err := IOSPOLL (723,response); {Performs a serial poll on
       device 723 and puts the response byte in RESPONSE.}
```

For C:

```
int     response;
int     error;

error = IOSPOLL(723L,&response); /*Perform a serial poll on
        device 723 and put the response byte in response.*/
```

Bus Activity

- ATN is set.
- UNL is sent.
- MLA is sent.
- TAD is sent.
- OSA is sent if specified.
- SPE is sent.
- ATN is cleared.
- Poll byte is read.
- ATN is set.
- SPD is sent.
- UNT is sent.

Comments

If a device is requesting service, it stops requesting service when its response byte is read.

Some devices are not capable of responding to a serial poll, in which case polling may result in an error. Consult the instrument manual to determine if an instrument can respond to a serial poll and how its response byte is interpreted.

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

IOSTATUS

This command determines the current interface status regarding a particular condition. It sets a variable representing that status.

Syntax

`IOSTATUS` (*select_code*, *condition*, *status_REF*)

select_code	specifies the interface select code.
condition	specifies the condition being checked, from 0 to 11. The possible conditions are:

Value	Description
0	Is the interface currently in the remote state? (always no)
1	What is the current state of the SRQ line?
2	What is the current state of the NDAC line?
3	Is the interface currently system controller?
4	Is the interface currently active controller?
5	Is the interface currently addressed as talker?
6	Is the interface currently addressed as listener?
7	What is the interface's bus address?
8	What is the state of the ATN line?
9	What is the address status of the interface?
10	What is on the DIO lines now?
11	What is the bus status of the interface?
12	What interfaced card is installed?

status_REF variable into which the condition's status is placed. It can have the following values:

Conditions 0 to 6 and 8

Value	Meaning
0	Clear or No
1	Set or Yes

Condition 7

Value	Meaning
0 to 30	Address of card

Condition 9*

Bit	Value	Meaning
0	1	ulpa
1	2	TADS
2	4	LADS
3	8	TPAS
4	16	LPAS
5	32	ATN
6	64	LLO
7	128	REM

Condition 10

Value	Meaning
0 to 255	Value of the data lines on the bus

Condition 11*

Bit	Value	Meaning
0	1	REN
1	2	IFC
2	4	SRQ
3	8	EOI
4	16	NRFD
5	32	NDAC
6	64	DAV
7	128	ATN

Condition 12

Value	Meaning
0	no interface card
1	HP 82990 (old)
2	HP 82335

* The actual value returned from conditions 9 and 11 will be the sum of the values of all true conditions. For example, the value returned if bits 2 and 3 were true would be 12.

To check whether a specific condition is true, use the AND operand in your language. For example, to check if DAV is true, you could call `IOSTATUS(7L,11,&result)`, then check whether (result AND 32) = 32, then (DAV is set). Make sure you are using the binary AND in your language and not the logical AND.

Examples

For Pascal:

```
VAR
condition : INTEGER;
   status : INTEGER;
      err : INTEGER;

 select := 1;
    err := IOSTATUS (7,condition,status);
           {Determine if SRQ is set.}
```

For C:

```
int     select;
int     status;
int     error;

select = 1;
error = IOSTATUS(7L,select,&status);
        /*Determine if SRQ is set.*/
```

Bus Activity

None.

Comments

Status conditions 9 through 11 are rarely used.

Possible errors are NOERR, ESEL, and ERANGE.

IOTAKECTL

This command takes active control from the currently active controller back to the HP-IB card.

Syntax

`IOTAKECTL` (*select_code*,*priority*)

select_code	specifies the interface select code.	
priority	specifies the priority of the request. This parameter can take one of three values:	
	1	Wait until the active controller passes control back to me. It will wait until it receives control or until it times out as specified by the IOTIMEOUT function.
	2	Assert SRQ with bits 1 and 6 set, then wait until the active controller passes control back to me. It will wait until either it receives control or until it times out as specified by the IOTIMEOUT function.
	3	Assert the Interface Clear (IFC) line. Asserting the IFC line immediately makes the HP 82335 the active controller. The HP 82335, however, *must* be the system controller to be able to assert the IFC line. If it is not the system controller, an ECTRL error will result.

Examples

For Pascal:

```
error  : INTEGER ;

error := IOTAKECTL(7, 1)
if error <> NOERR then writeln('an error occurred...');
```

For C:

```
int error ;

error = IOTAKECTL(7L, 1)
if (error != NOERR) printf ("an error occurred...\n");
```

Bus Activity

Bus activity is controlled by the active controller until IOTAKECTL is finished.

Comments

The Command Library defaults to address 30. If necessary, you can change this using the IOCONTROL command.

It may take awhile for the device that has active control to pass control back to the Command Library. You may want to increase your timeout value using IOTIMEOUT before calling IOTAKECTL, and decrease it after the IOTAKECTL call.

Possible errors are NOERR, ETIME, ESEL, ERANGE, and ECTRL.

IOTIMEOUT

This command sets an interface timeout value in seconds for I/O operations that do not complete (for example, the printer runs out of paper).

The default is timeout disabled.

Syntax

`IOTIMEOUT` (*select_code*, *timeout*)

select_code	specifies the interface select code.
timeout	specifies the length of the timeout period. A value of 0.0 disables the timeout, while a negative value results in an error.

Examples

For Pascal:

```
VAR
 timeout_val : REAL;
         err : INTEGER;

timeout_val := 1.23; {Timeout after 1.23 seconds.}
        err := IOTIMEOUT (7,timeout_val);
```

For C:

```
int       error;
double    timeout_val;

timeout_val = 1.23; /*Timeout after 1.23 seconds.*/
error = IOTIMEOUT(7L,timeout_val);
```

Bus Activity

None.

Comments

Timeout is effective for any interface operation that transfers data or commands.

A timeout error occurs only if timeout is enabled (that is, the timeout is set to a positive value).

Timeout should be established early in your program. It provides a way to recover from I/O operations that are not completed.

The timeout value is a real number specified in seconds, which gets rounded to the nearest 1/16 second. To timeout after 5 seconds, set timeout to 5.0. To timeout after 0.5 second, set timeout to 0.5. Note that a timeout of 0.0 effectively disables any timeouts. The maximum allowable timeout is 4096 seconds.

If a transfer times out, the incompleted transfer function returns the value 4, which corresponds to the ETIME error.

Possible errors are NOERR, ESEL, and ERANGE.

IOTRIGGER

This command triggers one or more devices.

Syntax

IOTRIGGER (*device_address*)
IOTRIGGER (*select_code*)

device_address	specifies a device address.
select_code	specifies the interface select code.

Examples

For Pascal:

```
VAR
 err : INTEGER;

err := IOTRIGGER (723);
```

For C:

```
int     error;

error = IOTRIGGER(723L);
```

Bus Activity

If a device address is specified:

- ATN is set.
- MTA is sent.
- UNL is sent.
- LAD is sent.
- OSA is sent if specified.
- GET is sent.

If a select code is specified:

- ATN is set.
- GET is sent.

Comments

Only one device can be triggered at a time if a device address is specified.

If a select code is specified, all devices on the bus that are addressed to listen (with IOSEND, for example) are triggered.

Possible errors are NOERR, ETIME, ECTRL, and ESEL.

Computer Museum

A

Error Descriptions

This appendix describes the Command Library errors. Error numbers depend upon the programming language, as shown in the first column of the table that follows.

Error Number BASIC / Pascal or C	**Mnemonic**	**Description**
0 / 0	NOERR	No error occurred.
100001 / 1	EUNKNOWN	Unknown error occurred. Check for malfunctioning equipment or incorrect hardware configuration. This error can also occur under these conditions: ■ IOENTERAB, IOENTERB. While using DMA, character matching must be disabled, and the swap size must be 1. ■ IOENTERS. While using DMA, character matching must be disabled. ■ IOOUTPUTAB, IOOUTPUTB. While using DMA, the swap size must be 1.

Error Number BASIC / Pascal or C	Mnemonic	Description
100002 / 2	ESEL	Invalid select code or device address was specified. This error would most likely occur under these conditions: ■ The first parameter of a call should have been a valid select code, but a device address or an invalid select code was specified instead. ■ A device address was expected, but a select code was given or a primary address outside the range 0 to 31 was specified. ■ The device address of the HP-IB interface was specified as a parameter in commands such as IOSPOLL, IOREMOTE, or IOCLEAR.

Error Number BASIC / Pascal or C	Mnemonic	Description
100003 / 3	ERANGE*	A command parameter was specified outside its allowable range. This error can occur under these conditions: ■ IOENTERA, IOENTERS. The specified length must be positive. ■ IOENTERAB, IOENTERB. The specified length must be positive. The swap size must be from 1 to 8. ■ IOEOL. The specified length must be from 0 to 8. ■ IOCONTROL. The specified value must be from 5 to 7. If 7 is selected, the valid address values are 0 to 30. ■ IOOUTPUTA, IOOUTPUTS. The specified length must be positive. ■ IOOUTPUTAB, IOOUTPUTB. The specified length must be positive. The swap size must be from 1 to 8. ■ IOPEN. The interrupt priority must be 0 or 1. ■ IOPPOLLC. The configuration value must be from 0 to 15. ■ IOSEND. The length must be positive. ■ IOSTATUS. The status specified was outside the range 0 to 7. ■ IOTIMEOUT. The specified timeout value must be greater than or equal to 0.

Error Number BASIC / Pascal or C	Mnemonic	Description
100004 / 4	ETIME	The time specified by IOTIMEOUT has elapsed since the last byte was transferred.
100005 / 5	ECTRL	The HP-IB interface must be the active controller or the system controller.
100006 / 6	EPASS	This error is obsolete.
100007 / 7	ENUM	Either no digit or an improperly formed number was received during real number input using IOENTER or IOENTERA. In this case, 0 is returned as the data value.
100008 / 8	EADDR	Improper talker or listener addressing occurred. An attempt was made to input or output data when the interface was not addressed to listen or talk. This error is likely to occur if a select code was specified instead of a device address, and the interface was not properly addressed to talk or listen.
100009 / 9	EFILE	A file error has occurred while reading, writing, or creating a file. This error could indicate either a disk full condition or a file does not exist for IOOUTPUTF.

* Potential conflict for C languages. See the following paragraph.

For C languages, ERANGE is defined to a different value by the MATH.H file. If MATH.H is included by a program, the compiler gives a warning and sets ERANGE to the last value defined. In this case, change one of the define statements so the names are different. For example, you could change two lines in the CHPIB.H Command Library file to

```
#define ERNGE 3
case ERNGE:    return (" Value out of range ");
```

and then use ERNGE for the Command Library error name (instead of ERANGE).

B

Summary of HP-IB

HP-IB Abbreviations

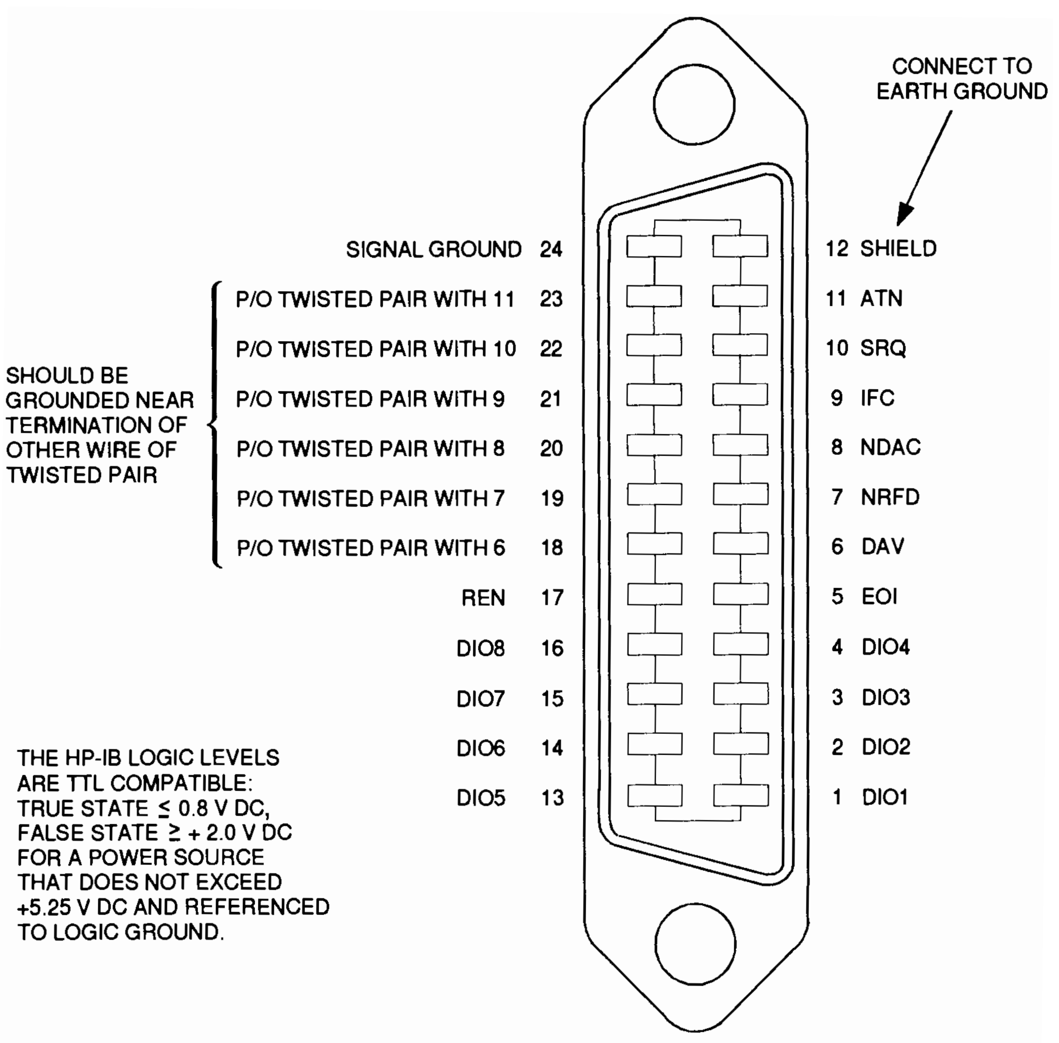
The following list defines the standard HP-IB abbreviations (mnemonics) used in this manual:

Mnemonic	**Definition**
ATN	Attention
DCL	Device Clear
EOI	End or Identify
EOL	End of Line
GET	Group Execute Trigger
GTL	Go To Local
IFC	Interface Clear
LAD	Listen Address
LLO	Local Lockout
MLA	My Listen Address
MTA	My Talk Address
OSA	Other Secondary Address
PPC	Parallel Poll Configure
PPD	Parallel Poll Disable
PPU	Parallel Poll Unconfigure
REN	Remote Enable
SDC	Selected Device Clear
SPD	Serial Poll Disable
SPE	Serial Poll Enable
SRQ	Service Request
TAD	Talk Address
TCT	Take Control
UNL	Unlisten
UNT	Untalk

HP-IB Description

The Hewlett-Packard Interface Bus (HP-IB) is HP's implementation of the IEEE-488 communication interface. It is used by a variety of instruments and peripherals manufactured by Hewlett-Packard and other companies. HP-IB is a 16-line bus that connects up to 15 devices in parallel on a communication link.

The following figure shows the HP-IB connector.


CONNECT TO EARTH GROUND
SIGNAL GROUND 24
P/O TWISTED PAIR WITH 11 23
P/O TWISTED PAIR WITH 10 22
P/O TWISTED PAIR WITH 9 21
P/O TWISTED PAIR WITH 8 20
P/O TWISTED PAIR WITH 7 19
P/O TWISTED PAIR WITH 6 18
SHOULD BE GROUNDED NEAR TERMINATION OF OTHER WIRE OF TWISTED PAIR
REN 17
DIO8 16
DIO7 15
DIO6 14
DIO5 13
12 SHIELD
11 ATN
10 SRQ
9 IFC
8 NDAC
7 NRFD
6 DAV
5 EOI
4 DIO4
3 DIO3
2 DIO2
1 DIO1

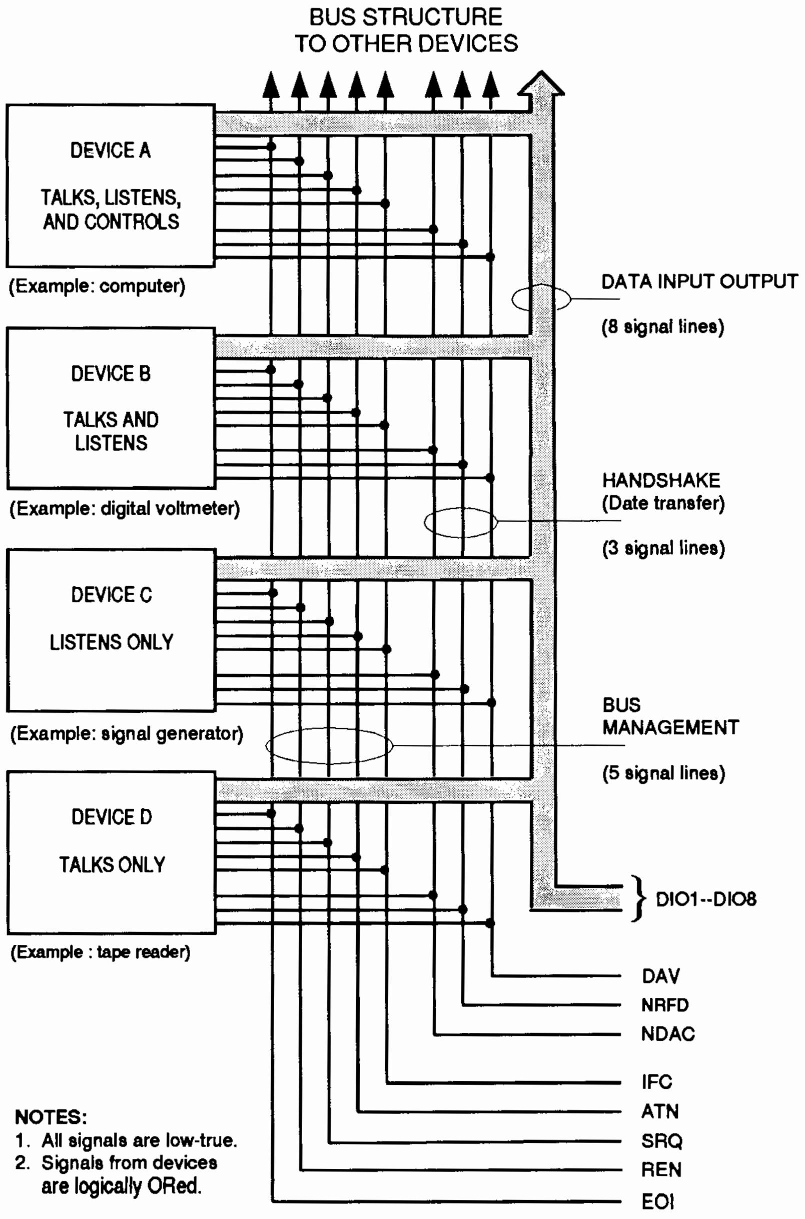
THE HP-IB LOGIC LEVELS ARE TTL COMPATIBLE:
TRUE STATE ≤ 0.8 V DC,
FALSE STATE ≥ + 2.0 V DC
FOR A POWER SOURCE THAT DOES NOT EXCEED +5.25 V DC AND REFERENCED TO LOGIC GROUND.

TYPE 57 MICRORIBBON CONNECTOR

Of the 16 signal lines, 8 are data lines, 3 are for handshake purposes, and the remaining 5 are control lines. Information is transferred across the eight data lines in a bit-parallel, byte-serial fashion. Briefly, the eight control and handshake lines are used as follows:

ATN	Attention is used primarily to differentiate between Command mode and Data mode. When ATN is true, information on the data lines is interpreted as a bus command; when ATN is false, the information is treated as a data byte.
EOI	End Or Identify has two uses. EOI is asserted on the last byte of a data transfer—this signals all listening devices that no more data should be expected on the transfer. EOI is used in combination with ATN to perform a parallel poll.
IFC	Interface Clear is under the exclusive control of the system controller. When it is pulsed true, all device interfaces are returned to an idle state and the state of the bus is cleared.
REN	Remote Enable may be set by the system controller to permit devices to operate in Remote mode—that is, under programmed HP-IB control instead of via the device's front panel.
SRQ	Service Request can be set by a device on the interface to indicate it is in need of service. SRQ might be set at the completion of a task such as taking a measurement, when an error is detected during device operation, or when requesting to be active controller.
DAV	Data Valid is a handshake line indicating that the active talker has placed data on the data lines (DIO1 through DIO8).
NRFD	Not Ready For Data is a handshake line indicating that one or more active listeners is not ready for more data, and the active talker should wait before sending new data on the bus.
NDAC	Not Data Accepted is a handshake line indicating that one or more active listeners has not accepted the current data byte, and the active talker should leave the current byte asserted on the data lines.

The following illustration shows the design of the bus.



NOTES:
1. All signals are low-true.
2. Signals from devices are logically ORed.

Commands and Data

There are two modes of communication on HP-IB: Command mode and Data mode.

In Command mode, information transmitted across the eight data lines is interpreted as talk or listen addresses, or universal address or unaddress commands (explained later). In this mode, only seven of the data lines are used. Some devices use the eighth line as a parity check for certain protocols.

In Data mode, any eight-bit value can be transmitted. The HP-IB can therefore be used for transmission of binary data as well as ASCII characters.

The three-line handshake scheme has several advantages. First, data transfer is asynchronous—the data rate is limited only by the speed of the devices actively involved in the transfer. A second, related advantage is that devices with different I/O speeds can be interconnected without need for other synchronization mechanisms. Also, multiple devices can be addressed concurrently.

Controllers, Talkers, and Listeners

To understand communication among devices, you should be familiar with the concepts of controller, talker, and listener.

Controller

Two types of controllers are defined within an HP-IB system: system controller and active controller.

There must be a single *system controller* capable of taking control of the interface at any time. The system controller has exclusive control over the IFC and REN lines.

Each system also has one or more devices capable of being *active controller* (sometimes referred to as controller-in-charge), although there may be only one active controller at any given time. The active controller has the ability to perform tasks such as establishing listeners and talkers, sending bus commands, and performing serial polls.

In most systems, a single computer will be both the system controller and the only active controller. Some non-system controller devices may request service indicating their desire to be active controller in order to perform some operation such as plotting data or directly accessing disk drives. The current active controller may "pass control" to a requesting device to make it the active controller. In other systems, a system controller may not be capable of operating as non-active controller, and therefore no pass-control capabilities will exist. Note that system controller capabilities may not be transferred.

An HP-IB system can be configured in one of three ways, and it affects the transfer of data as described:

- No controller. This mode of data transfer is limited to a direct transfer between one device manually set to talk only, and one or more devices manually set to listen only.
- Single controller. In this configuration, data transfer can be from controller to devices (Command or Data mode), from a device to controller (Data mode only), or from a device to other devices (Data mode only).
- Multiple controllers. This mode of data transfer is similar to that of a single controller, with the requirement that active controller status be passable from one controller to another. In this configuration, one controller must be designated as the system controller. This controller is the only one that can control the IFC and REN lines.

  Control is passed to another controller by addressing it as a talker and commanding it to "take control" (TCT).

Computer Museum

Talker

In each system there can be at most one device addressed as talker at any given time. A device becomes addressed as talker by receiving its talk address from the active controller. Each device on the bus must have a unique bus address. This address is usually set at the manufacturing site, but it may be set by switches on the instrument.

The addresses are in the range 0 to 30. A talk address is formed by adding the primary bus address to the talk address base value of 64 and transmitting that value across the data lines while ATN is asserted. For example, talk address 9 would be formed by asserting ATN and transmitting a byte whose value is 73 (64 + 9 = 73, ASCII character "I").

Listener

Listen addresses are formed in a similar manner to talk addresses, except that listen addresses use a base of 32. For example, listen address 9 is sent as value 41 transmitted with ATN true (32 + 9 = 41, ASCII character ")").

Multiple devices may be addressed to listen at any time, and data bytes will be received by all listeners in parallel. However, most devices cannot be addressed to both talk and listen at the same time. (See the table at the end of this appendix for talk and listen address codes.)

Extended Addressing

The descriptions of talk address and listen address refer to a device's primary address. Some devices also have extended talker or extended listener capabilities, sometimes used as secondary addresses or as device-dependent commands. With extended addressing, talk and listen addresses are represented by two command bytes. The first byte is the primary talk or listen address as described above. The second byte is a secondary address command.

Secondary addresses may be in the range 0 to 31. The secondary commands transmitted are formed by adding the secondary address to the base value 96 and transmitting the byte with ATN true.

Extended addresses can be used, for example, to access a specific I/O card within an instrument that allows multiple I/O cards.

Bus Commands

Five types of information are transmitted when the bus is operating in Command mode (that is, when ATN is asserted):

- Talk addresses.
- Listen addresses.
- Universal commands.
- Addressed commands.
- Unaddress commands.

Talk addresses and listen addresses are discussed above. The other categories are described below.

Universal Commands

Universal commands are received by all responding devices on the bus whether addressed to listen or not. The commands are listed below.

Mnemonic	Command	Description
LLO	Local Lockout	Disables the front panel of the responding device. The REN line must be asserted in order for LLO to have any effect. If the instrument is already in Remote mode, the lockout will be immediate. Otherwise, the lockout will commence when the device receives its listen address.
DCL	Universal Device Clear	All devices capable of responding are returned to some known, device-dependent state. In some cases a device will perform a self-test in response to a Universal Device Clear.
PPU	Parallel Poll Unconfigure	Directs all devices on the HP-IB that have parallel poll configure capabilities to not respond to a parallel poll.
SPE	Serial Poll Enable	Enables Serial Poll mode on the interface.
SPD	Serial Poll Disable	Disables Serial Poll mode on the interface.

Addressed Commands

Addressed commands are executed only by those devices that are currently addressed as listeners. They allow the controller to initiate a simultaneous action by a selected group of devices on the bus, such as triggering them to take readings at the same time. The commands are listed below.

Mnemonic	Command	Description
SDC	Selected Device Clear	Similar to a Universal Device Clear (DCL) with only those devices addressed to listen responding.
GTL	Go To Local	Returns devices that are addressed to listen to Local mode (re-enables front panel programming). REN stays asserted when a GTL is sent, and devices will be returned to Remote upon receipt of their listen address.
GET	Group Execute Trigger	Initiates some preprogrammed action by listening devices. This may be used to simultaneously start action in a group of devices that are addressed to listen.
PPC	Parallel Poll Configure	Configures a device to respond to a parallel poll on a specified data line with either a positive or negative signal. A secondary command sent after PPC contains the data that configures the device.
TCT	Take Control	Transfers active controller status to the device that is currently addressed to talk.

Unaddress Commands

The two unaddress commands can be considered as extensions of talk and listen addresses.

UNL (Unlisten) causes all devices on the bus (except those that have a built-in switch set to Listen Only) to stop being listeners. UNL is equivalent to listen address 31.

UNT (Untalk) directs any device on the interface to no longer be addressed as talker. Since there may only be one device addressed to talk at any time, receipt of another device's talk address is equivalent to receiving a UNT. UNT is equivalent to talk address 31.

Service Requests

Some devices that operate on the interface have the ability to request service from the system controller. A device may request service when it has completed a measurement, when it has detected a critical condition, or under many other circumstances.

A service request (SRQ) is initiated when the device sets the SRQ line true. The controller, sensing that SRQ has been set (typically either by polling the status of the line or by enabling an SRQ interrupt), can poll devices in one of two ways: serial poll or parallel poll.

Serial Poll

A typical sequence of events in performing a serial poll is:

- Establish a device as a talker.
- Send SPE to set up Serial Poll mode.
- Wait for the addressed device to send its serial poll response byte.
- Send an SPD and UNT to disable the Serial Poll mode.

The meaning of the serial poll response byte depends upon the individual device. However, if bit 6 of the response byte (bit value 64) is 1, the device is indicating it has requested service. If bit 6 is 0, the polled device is not the one that requested service. Individual device manuals provide additional information on the meanings of serial poll response bytes.

Parallel Poll

Parallel polling permits the status of multiple devices on HP-IB to be checked simultaneously. Each device is assigned a data line (DIO1 through DIO8) that the device sets true during the parallel poll routine if it requires service.

More than one device can be assigned to a particular data line. If a shared line is sensed true, a serial poll can typically be performed to determine which device set the line. A parallel poll is started when the controller asserts ATN and EOI together. After a short period of time the controller reads the poll byte and begins its interpretation thereof.

Some devices can be configured (by the PPC command) to respond on specific data lines. Other devices may respond on lines selected by switches or jumpers in the devices. Some devices do not have parallel poll capability.

ASCII Codes

The following table lists ASCII codes, characters, and corresponding HP-IB commands. You can used these characters with the IOSEND Library command to send HP-IB commands.

Code	Char	Cmd	Code	Char	Cmd	Code	Char	Cmd	Code	Char	Cmd
0	NUL		32	SP	L0	64	@	T0	96	‘	
1	SOH	GTL	33	!	L1	65	A	T1	97	a	
2	STX		34	"	L2	66	B	T2	98	b	
3	ETX		35	#	L3	67	C	T3	99	c	
4	EOT	SDC	36	$	L4	68	D	T4	100	d	
5	ENQ	PPC	37	%	L5	69	E	T5	101	e	
6	ACK		38	&	L6	70	F	T6	102	f	
7	BEL		39	’	L7	71	G	T7	103	g	
8	BS	GET	40	(	L8	72	H	T8	104	h	
9	HT	TCT	41	)	L9	73	I	T9	105	i	
10	LF		42	*	L10	74	J	T10	106	j	
11	VT		43	+	L11	75	K	T11	107	k	
12	FF		44	,	L12	76	L	T12	108	l	
13	CR		45	–	L13	77	M	T13	109	m	
14	SO		46	.	L14	78	N	T14	110	n	
15	SI		47	/	L15	79	O	T15	111	o	
16	DLE		48	0	L16	80	P	T16	112	p	
17	DC1	LLO	49	1	L17	81	Q	T17	113	q	
18	DC2		50	2	L18	82	R	T18	114	r	
19	DC3		51	3	L19	83	S	T19	115	s	
20	DC4	DCL	52	4	L20	84	T	T20	116	t	
21	NAK	PPU	53	5	L21	85	U	T21	117	u	
22	SYN		54	6	L22	86	V	T22	118	v	
23	ETB		55	7	L23	87	W	T23	119	w	
24	CAN	SPE	56	8	L24	88	X	T24	120	x	
25	EM	SPD	57	9	L25	89	Y	T25	121	y	
26	SUB		58	:	L26	90	Z	T26	122	z	
27	ESC		59	;	L27	91	[	T27	123	{	
28	FS		60	<	L28	92	\	T28	124	|	
29	GS		61	=	L29	93	]	T29	125	}	
30	RS		62	>	L30	94	∧	T30	126	~	
31	US		63	?	UNL	95	_	UNT	127	DEL	

Reserved Names

The following pages list reserved names that are used internally by the Command Library. You should avoid using these names in your programs.

GW BASIC, Vectra BASIC, and BASICA:		
(No reserved names for the Command Library)		
QuickBASIC 4.0 and later, Microsoft Compiled BASIC, and BASIC PDS:		
`DEFERR`	`HPIBLIB_PEN`	`PENRESTORE`
`PENSETUP`		
(Plus next page)		
Pascal Languages:		
(See next page)		
C Languages:		
(See next page)		

All Languages (except GW BASIC, Vectra BASIC, and BASICA):

A_GETBYTE	HPIBLIB_IODMA	HPIBLIB_SENDCMD
HPIBLIB_ABORT	HPIBLIB_LISTENCTL	HPIBLIB_SENDSTR
HPIBLIB_ADDRESS	HPIBLIB_LOCAL	HPIBLIB_SETADDR
HPIBLIB_ATNCTL	HPIBLIB_LOCKOUT	HPIBLIB_SETTIMEOUT
HPIBLIB_CHECKADDR	HPIBLIB_MATCH	HPIBLIB_SHORTT1
HPIBLIB_CHECKLINES	HPIBLIB_OUTPUTA	HPIBLIB_SPOLL
HPIBLIB_CHECKSTATE	HPIBLIB_OUTPUTAB	HPIBLIB_STATUS
HPIBLIB_CHIPRESET	HPIBLIB_OUTPUTB	HPIBLIB_TAKECTL
HPIBLIB_CLEAR	HPIBLIB_OUTPUTF	HPIBLIB_TALKCTL
HPIBLIB_CONTROL	HPIBLIB_OUTPUTS	HPIBLIB_TERMREASON
HPIBLIB_DMAREAD	HPIBLIB_PASSCTL	HPIBLIB_TIMEOUT
HPIBLIB_DMAWRITE	HPIBLIB_PPOLL	HPIBLIB_TRIGGER
HPIBLIB_DOPPOLL	HPIBLIB_PPOLLC	HPIBLIB_GETFILE
HPIBLIB_ENTERA	HPIBLIB_PPOLLU	HPIBTOOL_SYSCTL
HPIBLIB_ENTERAB	HPIBLIB_READREG	HPLIBGET
HPIBLIB_ENTERB	HPIBLIB_RELEASERFD	HPLIBPUT
HPIBLIB_ENTERF	HPIBLIB_REMOTE	INSTVECT
HPIBLIB_ENTERS	HPIBLIB_RENCTL	ISCTERMREASON
HPIBLIB_EOICTL	HPIBLIB_REQUEST	NEW_IRQ
HPIBLIB_EOL	HPIBLIB_RESET	PASSRESTORE
HPIBLIB_FASTOUT	HPIBLIB_SEND	PASSSETUP
HPIBLIB_GETBIN	HPIBLIB_SENDFILE	REQUEST
HPIBLIB_GETBYTE	HPIBLIB_SENDBIN	RESTOREVECT
HPIBLIB_GETSTR	HPIBLIB_SENDBYTE	WAITTCT
HPIBLIB_IOABORT		

Index

Computer Museum

HEWLETT
PACKARD

Customer Reorder Number
82335-90006

Printed in U.S.A.   E1291

82335-90606   Manufacturing Number