NI-488.2™ Software Reference Manual for MS-DOS

June 1997 Edition

Part Number 370889A-01

Internet Support
`support@natinst.com`
E-mail: `info@natinst.com`
FTP Site: `ftp.natinst.com`
Web Address: `http://www.natinst.com`

Bulletin Board Support
BBS United States: (512) 794-5422
BBS United Kingdom: 01635 551422
BBS France: 01 48 65 15 59

Fax-on-Demand Support
(512) 418-1111

Telephone Support (U.S.)
Tel: (512) 795-8248
Fax: (512) 794-5678

International Offices
Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 5734815, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 520 2635, Netherlands 0348 433466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200, U.K. 01635 523545

National Instruments Corporate Headquarters
6504 Bridge Point Parkway Austin, TX 78730-5039
Tel: (512) 794-0100

Copyright

Trademarks

Warning Regarding Medical and Clinical Use of National Instruments Products

Preface

This manual describes in detail the NI-488.2 for MS-DOS software. All NI-488.2 routines and NI-488 functions for Advanced IBM Interpretive BASIC for the IBM Personal Computer, Microsoft QuickBASIC, Microsoft Professional BASIC, and Microsoft C are described. Software and manual supplements are available for use with other programming languages.

Organization of This Manual

This manual is organized as follows:

- Chapter 1, *Introduction*, introduces you to the General Purpose Interface Bus (GPIB) and this manual.
- Chapter 2, *NI-488.2 Software Installation and Configuration*, contains the instructions for installing and configuring the NI-488.2 software. It also contains a description of `IBCONF`, the software configuration program, and a description of NI-488.2 software features helpful in developing and debugging your application program.
- Chapter 3, *Understanding the NI-488.2 Software,* introduces you to the NI-488.2 routines and NI-488 functions, describes features common to both, and contains general programming information you need to know before starting your application program. It also contains specific programming information for each language.
- Chapter 4, *NI-488.2 Software Characteristics and Routines*, contains a discussion of the important characteristics of the NI-488.2 routines available in the MS-DOS driver that are common to all programming languages. It also contains a detailed description of each NI-488.2 routine with example programs. The descriptions are listed alphabetically for easy reference.
- Chapter 5, *NI-488 Software Characteristics and Functions,* contains a discussion of the important characteristics of the NI-488 functions available in the MS-DOS driver that are common to all programming languages. It also contains a detailed description of each NI-488 function with example programs. The descriptions are listed alphabetically for easy reference.

- Chapter 6, *IBIC*, contains instructions for running the Interface Bus Interactive Control (`IBIC`) program using the NI-488.2 routines and NI-488.2 functions. It also contains the syntax of these functions, important characteristics, and `IBIC` example programs.
- Chapter 7, *Applications Monitor*, explains how to use, install, and configure the applications monitor, a resident program that is useful in debugging sequences of GPIB calls from within your application.
- Appendix A, *Multiline Interface Messages*, contains a multiline interface message reference list, which describes the mnemonics and messages that correspond to the interface functions.
- Appendix B, *Common Errors and Their Solutions*, lists the most common errors and their solutions. They are arranged according to the error code that is returned from the function as indicated by `iberr`. This appendix also describes error situations that do not return an error code.
- Appendix C, *Serial/Parallel Port Redirection to the GPIB*, describes how to use standard DOS printer commands (LPRINT, PRINT, and so on) to redirect data from the serial or parallel port to a GPIB printer, plotter, or other device using the NI-488.2 for MS-DOS driver.
- Appendix D, *Operation of the GPIB*, describes some basic concepts you should understand to operate the GPIB. It also contains a description of the physical and electrical characteristics as well as configuration requirements of the GPIB.
- Appendix E, *Customer Communication*, contains forms for you to complete to facilitate communication with National Instruments concerning our products.
- The *Glossary* contains an alphabetical list of terms used in this manual and a definition of each.
- The *Index* contains an alphabetical list of key terms and topics used in this manual, including the page where each one can be found.

Conventions Used in This Manual

Throughout this manual, the following conventions are used to distinguish elements of text:

italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept.
`monospace`	Lowercase text in this font denotes text or characters that are to be literally input from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, variables, filenames, and extensions, and for statements and comments taken from program code.
`bold monospace`	Bold lowercase text in this font denotes the messages and responses that the computer automatically prints to the screen.
`italic monospace`	Italic lowercase text in this font denotes that you must supply the appropriate words or values in the place of these items.
<>	Angle brackets enclose the name of a key on the keyboard–for example, <PageDown>.
-	A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys–for example, <Control-Alt-Delete>.
<Enter>	Key names are capitalized.

Definitions of Terms Used in This Manual	
Term	**Reference**
BASICA	Advanced IBM Interpretive BASIC for the IBM Personal Computer
QuickBASIC	Microsoft QuickBASIC
BASIC	Microsoft Professional BASIC
C	Microsoft C

Abbreviations

The following metric system prefixes are used with abbreviations for units of measure in this manual.

Prefix	**Meaning**	**Value**
m-	milli-	10^{-3}
µ-	micro-	10^{-6}
n-	nano-	10^{-9}

The following are abbreviations used in this manual:

dec	decimal
hex	hexadecimal
Hz	hertz
m	meters
M	megabytes of memory
oct	octal
sec	seconds

Acronyms

The following acronyms are used in this manual:

AC	alternating current
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
CPU	central processing unit
DMA	direct memory access
DVM	digital voltmeter
EOI	End-or-Identify
EOS	End-of-String
FIFO	first-in-first-out
GPIB	General Purpose Interface Bus
IBIC	Interface Bus Interactive Control
IEEE	Institute of Electrical and Electronic Engineers
I/O	input/output
ISO	International Standards Organization
PC	personal computer
RAM	random-access memory
SCPI	Standard Commands for Programmable Instruments
TTL	transistor-transistor logic
ULI	universal language interface
VAC	volts alternating current

Note: References in this manual to IEEE-488 and IEEE-488.2 are referring to the ANSI/IEEE Standard 488.1-1987 and the ANSI/IEEE Standard 488.2-1987, respectively, which define the GPIB.

Related Documentation

The following documents contain information that you may find helpful as you read this manual:

- ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation*

- ANSI/IEEE Standard 488.2-1987, *IEEE Standard Codes, Formats, Protocols, and Common Commands*

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix E, *Customer Communication*, at the end of this manual.

Contents

Figures

Tables

Chapter 1
Introduction

This chapter introduces you to the General Purpose Interface Bus (GPIB) and this manual. The National Instruments family of GPIB products for your personal computer will become a valued and integral part of your work environment.

Introduction to the GPIB

The GPIB is a link, or interface system, through which interconnected electronic devices communicate.

History of the GPIB

The original GPIB was designed by Hewlett-Packard (where it is called the HP-IB) to connect and control programmable instruments manufactured by Hewlett-Packard. Because of its high data transfer rates of up to 1 megabyte/sec, the GPIB quickly gained popularity in other applications such as intercomputer communication and peripheral control. It was later accepted as the industry standard IEEE-488. The versatility of the system prompted the name General Purpose Interface Bus.

National Instruments expanded the use of the GPIB among users of computers manufactured by companies other than Hewlett-Packard. National Instruments specializes both in high-performance, high-speed hardware interfaces and in comprehensive, full-function software that helps users bridge the gap between their knowledge of instruments and computer peripherals and of the GPIB itself.

Background

This manual was developed as part of the documentation for the NI-488.2 for MS-DOS software. Software reference material can be found in this manual. Hardware-specific information can be found in the documentation provided with hardware items.

The terms NI-488.2 for MS-DOS, NI-488 for MS-DOS, and GPIB-PC are not analogous. GPIB-PC is the original name for the family of NI-488 GPIB Controller software used specifically for MS-DOS personal computers; while NI-488 for MS-DOS is the generic and preferred name for the same family of NI-488 GPIB Controller software for personal computers. NI-488.2 for MS-DOS is an enhanced version of the two earlier products and only runs on interface boards that are 100% compliant with IEEE-488.2.

Chapter 2
Installation and Configuration of NI-488.2 Software

This chapter contains instructions for installing and configuring the NI-488.2 software. It also contains a description of `IBCONF`, the software configuration program, and a description of NI-488.2 software features helpful in developing and debugging you application program.

The NI-488.2 Software Package

Before installing the software, you should understand the files that will be copied from the distribution diskette and what their purpose is.

Main Programs and Files

The following programs and files are required to program the GPIB interface from BASICA, QuickBASIC, Microsoft BASIC, or Microsoft C using the NI-488.2 software:

- `GPIB.COM` is the NI-488.2 driver that is loaded at system startup by the Disk Operating System.
- `BIB.M` is a binary language interface file that gives an application program written in BASICA access to the NI-488.2 software.
- `QBIB.OBJ` is a binary language interface file that gives an application program written in QuickBASIC (version 4.0 or higher) access to the NI-488.2 software.
- `MBIB.OBJ` is a binary language interface file that gives an application program written in Microsoft Professional BASIC (version 7.0) access to the NI-488.2 software.
- `MCIB.OBJ` is a binary language interface file that gives an application program written in Microsoft C (version 4.0 or higher) access to the NI-488.2 software.

- `DECL.BAS` is a declaration file that contains code you should place at the beginning of BASICA application programs.
- `QBDECL.BAS` is a declaration file that contains code you should place at the beginning of QuickBASIC application programs.
- `MBDECL.BAS` is a declaration file that contains code you should place at the beginning of Microsoft BASIC application programs.
- `DECL.H` is a header file for use with Microsoft C applications.

Additional Programs and Files

The following programs and files are used to install and test the NI-488.2 software, diagnose hardware problems, and provide examples for programming from BASICA, QuickBASIC, BASIC, and C:

- `IBDIAG.EXE` is a program that you can use to test the hardware installation. The installation program uses `IBDIAG.EXE` to confirm that the hardware is functioning properly.
- `IBTEST.EXE` is a program that you can use to test the NI-488.2 software.
- `INSTALL.EXE` is a multipurpose, menu driven program that you can use to install the NI-488.2 software and test the hardware and software configuration. When `INSTALL.EXE` installs the NI-488.2 software, it updates your `CONFIG.SYS` (the MS-DOS system configuration file).
- `IBCONF.EXE` is a software configuration program. It can be used to change the configuration of the NI-488.2 software.
- `IBIC.EXE` is an interactive control program that executes NI-488.2 functions that you enter from the keyboard. It helps you learn the functions, program instruments or other GPIB devices, and develop your application program.
- `APPMON.EXE` is the applications monitor program. It is a memory resident program that is useful in debugging your application. The applications monitor can halt program execution (trap) on return from GPIB software calls, so that you can inspect function arguments, buffers, return values, GPIB global status variables, and other pertinent data. The applications monitor performs automatic error detection.

- `IBTRAP.EXE` is a program that configures the applications monitor.
- `ULI.COM` is the Universal Language Interface software file needed to use the Universal Language Interface option of the NI-488.2 software package.
- `DBSAMP.BAS`, `DQBSAMP.BAS`, `DMBSAMP.BAS`, `DCSAMP.C`, and `DIBSAMP` are example programs for device calls in BASICA, QuickBASIC, BASIC, C, and `IBIC`. `BBSAMP.BAS`, `BQBSAMP.BAS`, `BMBSAMP.BAS`, `BCSAMP.C`, and `BIBSAMP` are example programs for board calls in BASICA, QuickBASIC, BASIC, C, and `IBIC`.
- `BSAMP488.BAS`, `QBSAMP488.BAS`, `MSAMP488.BAS`, `CSAMP488.C`, and `SAMP488` are example programs for 488.2 calls in BASICA, QuickBASIC, BASIC, C, and IBIC. Additional examples may be found in Chapter 4, *NI-488.2 Software Characteristics and Routines*, of this manual.

Readme files are included on the distribution diskette. `README.DOC` discusses the NI-488.2 software. The remaining Readme files discuss programming considerations for the supported languages.

Step 1. Installing the Software

The NI-488.2 distribution diskette contains a program named `INSTALL.EXE` that installs and tests the NI-488.2 software for you. The term *source disk* or *source directory* refers to the NI-488.2 distribution diskette. The term *destination directory* refers to the location on your hard disk or diskette where the software will be installed (usually `c:\boardname` (for example, `c:\AT-GPIB`)). The term *boot drive* refers to the drive that is read by your computer when you power-on or restart your computer.

Preparation

Your first step depends on whether you start up your computer from a diskette or a hard disk. Perform the step that applies to your system.

- **Starting from a Diskette**

  If you start MS-DOS from a diskette, your destination diskette must have enough free space to hold a copy of the NI-488.2 software that is located on the distribution diskette. The `README.DOC` file on the distribution diskette contains a list of the amount of disk space needed to install each part of the NI-488.2 software.

Power-on your system if you have not already done so. Insert your system diskette into the primary disk drive (usually named A:) and the distribution diskette into the secondary drive (usually named B:).

- **Starting from a Hard Disk**

  If you start MS-DOS from a hard disk, the hard disk must have enough free space to hold a copy of the NI-488.2 software that is located on the distribution diskette. The `README.DOC` file on the distribution disk contains a list of the amount of disk space needed to install each part of the NI-488.2 software.

  If you have a previous version of the NI-488 software loaded on your system, you should remove the old files before installing the new ones.

  Power-on your system if you have not already done so. Then insert the NI-488.2 distribution diskette into a diskette drive.

Run INSTALL

After starting the computer, run `INSTALL` from the NI-488.2 distribution diskette by first entering:

```
y:install                    <Enter>
```

where `y` is the name of the disk drive containing the distribution diskette.

`INSTALL` brings up with a main menu of four options: `Partial GPIB Installation`, `Full GPIB Installation`, `Diagnostics`, and `Return to DOS`.

- Selecting `Partial GPIB Installation` brings up a menu where you can select which parts of the NI-488.2 software to install. After selecting these items, the installation screen appears. If you are low on free disk space, you will probably want to use this option.

- Selecting `Full GPIB Installation` immediately takes you to the installation screen. After you complete this screen, the main menu appears.

- By selecting the third option on the main menu, `Diagnostics`, you can run the hardware and/or software diagnostics programs.
- The final option, `Return to DOS`, exits the install program.

Note: You may press <Escape> at any time to abort the installation.

Partial Installation

Selecting `Partial GPIB Installation` causes the `INSTALL` program to bring up another menu. This menu contains six options, any or all of which can be selected. The following list describes these options:

- Driver and Support Files. This option copies the NI-488.2 software (`GPIB.COM`) and its supporting programs (`IBIC.EXE`, `IBCONF.EXE`, `APPMON.EXE`, `IBTRAP.EXE`, `IBTEST.EXE`, `IBDIAG.EXE`, `IBTSTB.EXE`, `BIBSAMP`, `DIBSAMP`, `SAMP488`, and `README.DOC`).
- Microsoft C Language Interface. This option copies the files needed to interface a Microsoft C application program to the NI-488.2 software.
- BASIC 7.X Language Interface. This option copies the files needed to interface a Microsoft BASIC 7.X application program to the NI-488.2 software.
- QuickBASIC Language Interface. This option copies the files needed to interface a Microsoft QuickBASIC application program to the NI-488.2 software.
- BASICA Language Interface. This option copies the files needed to interface a BASICA application program to the NI-488.2 software.
- Universal Language Interface. This option copies the files needed to interface an application program to the Universal Language interface of the NI-488.2 software.

You can select any or all of these options by using the arrow keys and the space bar. When you have completed your selections, press <Enter> to continue. The `INSTALL` program now displays the installation screen.

Full Installation

INSTALL displays a screen for you to confirm or change the default installation settings. There are three fields on the screen, each with a default value that INSTALL has determined to be appropriate for your system. If you need to change any of these options, use the arrow keys to move around on the screen. Press <Enter> to begin the installation.

The three pieces of information needed by INSTALL are: the *source directory*, the *destination directory*, and the *boot drive*.

- The Destination Directory. This is the drive and directory name of the location to which INSTALL copies the NI-488.2 software. If you are installing the software onto a hard disk, this field typically contains C:\boardname (for example, C:\AT-GPIB)). In most cases, the drive name given in this field should be the drive from which you start your computer.

- The Boot Drive. This field should contain the drive name of the disk (hard disk or diskette) from which you start your computer. If you start from a diskette, this field typically contains A:. If you start from a hard disk, this field typically contains C:.

After you have confirmed the above information, press <Enter>. INSTALL then checks the information you entered. If there is a problem accessing any of the specified drives, it displays an error message and prompts you to re-enter the offending field.

When the installation is complete, INSTALL asks to modify your CONFIG.SYS file. If you enter yes, INSTALL adds the following line to your CONFIG.SYS file:

```
device=dir\gpib.com
```

where dir is the directory to which INSTALL copied the NI-488.2 software files; for example, if you are using the AT-GPIB, dir may be C:\AT-GPIB. If you have a previous version of the NI-488 software installed on your computer, INSTALL replaces the information in the old version file CONFIG.SYS with the new information.

If you enter no to the above prompt, INSTALL displays a message informing you of the correct line that you should add to your CONFIG.SYS file.

Diagnostics

Upon completion of the software installation, `INSTALL` returns to the main menu. If you have already installed your GPIB interface board into your computer, select the `Diagnostics` menu. If you have not installed your GPIB interface board, press <Escape> to exit `INSTALL`. Follow the hardware installation instructions given in the Getting Started manual that you received with your board to install your interface board. After installing your interface board, power-on your computer and rerun `INSTALL`. Select the `Diagnostics` menu and complete the following instructions.

From the `Diagnostics` menu you have the choice of running the hardware diagnostics (`IBDIAG.EXE`), the software diagnostics (`IBTEST.EXE`), or both. Select the hardware diagnostics. If `IBDIAG` returns an error message, check the following items:

- Check the hardware configuration to ensure that the switch and jumper settings are correct:
  - The hardware configuration settings are the same as the values you entered when prompted by the `IBDIAG` program.
  - The hardware configuration settings should not configured to the same setting as another board or device in your computer. Try reconfiguring the hardware and run `IBDIAG` again.
- Check the GPIB interface board to ensure that it is not connected to a GPIB device. `IBDIAG` requires that the interface board *not* be connected to any GPIB devices.

If you have performed these steps and there is still a problem, complete the Hardware and Software Configuration Form in the Appendix of the Getting Started manual that you received with your interface board and then contact National Instruments for technical support.

If the hardware diagnostics passed, you can proceed through the remainder of this chapter.

Step 2. Software Configuration

Before you can run the software diagnostics tests, the NI-488.2 software must be loaded. If you have just completed the installation procedure and have not restarted your computer, the software is not yet loaded. Exit `INSTALL` by pressing <Escape>. Before restarting your computer, you may need to run `IBCONF` (the software configuration program).

You must run `IBCONF` if you have made any changes to the GPIB hardware switches or jumper settings or you need to change any of the default software configuration options. If you have not changed the hardware in any way and the default software configuration is acceptable, there is no need to run `IBCONF`. You can, however, run `IBCONF` just to examine the software configuration if you want.

Refer to the `IBCONF` section later in this chapter for information on how to run `IBCONF` and on the configurable software options and their default values.

Step 3. Software Diagnostics

To load the NI-488.2 software into the memory of the computer, you must restart the computer. (If you start up your computer from a diskette, use that same diskette to restart your computer.) You will typically only have to do this once when you first install the NI-488.2 software and whenever you need to reconfigure the hardware settings.

Restart your computer by pressing <Control-Alt-Delete>. This restarts the computer and loads the NI-488.2 software into memory.

After the NI-488.2 software is installed, run `IBTEST`. Running `IBTEST` ensures that the NI-488.2 software is installed properly on your system. If an error occurs, check the following:

- The GPIB interface board should *not* be connected to a GPIB device. `IBTEST` requires that the GPIB interface board not be connected to any GPIB devices.

- If you changed any of the hardware configurations on the GPIB interface board, check the current software configuration of the software by running `IBCONF`. Make sure that the hardware settings match. Refer to the *Step 2. Software Configuration* section earlier in this chapter.

- Has the `CONFIG.SYS` file on your startup disk been correctly modified to contain the following line?

  ```
  device=dir\gpib.com
  ```

  where `dir` is the directory to which `INSTALL` copied the NI-488.2 software files; for example, if you are using the AT-GPIB, `dir` may be `C:\AT-GPIB`.

- Did you restart your computer after installing and configuring the NI-488.2 software? Refer to the *Step 2. Software Configuration* section earlier in this chapter.

If you have performed these steps and there is still a problem, complete the Hardware and Software Configuration Form in the Appendix of the Getting Started manual that you received with your interface board and then contact National Instruments for technical support.

If no errors occurred, you can proceed to learn how to use the software and how to develop your application program.

Step 4. Developing Your Application Program

The easiest way to learn to communicate with your instrument is by controlling it interactively. Located in your GPIB directory is an interactive control program entitled `IBIC.EXE`. With this program you can write to and read from your instrument while it updates the status of these GPIB transfers and displays any errors that occur.

To use this program, connect your instrument to the bus and enter the following commands:

```
cd \dir
```

where `dir` is the name of the directory to which `INSTALL` copied the NI-488.2 software.

```
IBIC
```

Now begin entering functions as described in Chapter 6, *IBIC*, later in this manual.

IBCONF

`IBCONF.EXE` is a screen-oriented, interactive program that is used to modify the configuration parameters for your interface board and the GPIB devices connected to it. It has a supplementary batch mode which can be used for configuring in a non-interactive fashion.

When used interactively, `IBCONF` reads in the configuration parameters from a GPIB driver file on your disk and displays them for your inspection. You can alter any of the parameters to suit your requirements. Once you have finished modifying the configurable parameters, these changes can be saved into the GPIB driver file on disk when you exit the `IBCONF` program.

There are two methods by which the changes made in `IBCONF` to the GPIB driver can be reflected in the memory-resident GPIB driver used by DOS applications. The traditional method is to restart your computer so that DOS can reload the modified GPIB device driver into memory. A second, easier method, is to let `IBCONF` modify the memory-resident GPIB driver when you exit `IBCONF`. This second method works only if the two copies of the GPIB driver (the one stored on disk and the one loaded into memory by DOS) are compatible.

The simplest way to use `IBCONF` is to change to the directory that contains the installed GPIB distribution files and enter the following command:

```
IBCONF
```

If the current directory does not contain the driver, `GPIB.COM`, `IBCONF` searches for a `GPIB.COM` to modify. In general, `IBCONF` finds a `GPIB.COM` file to configure by going through the following process:

1. If the file `c:\config.sys` exists and contains a line of the format `device=<path>gpib.com`, that `GPIB.COM` file is configured.
2. If the file `config.sys` exists on the root directory of the current drive and contains a line of the format `device=<path>gpib.com`, that `GPIB.COM` file is configured.
3. If a `GPIB.COM` file exists in the current directory, that file is configured.

Table 2-1 lists the options with which `IBCONF` can be started.

Table 2-1. IBCONF Options

IBCONF Option	**Action**
`driver`	`IBCONF` will configure the given driver file instead of searching for a `gpib.com` file (for example, `IBCONF d:\at-gpib\at-gpib.com`).
`-b filename`	Batch mode configuration. `IBCONF` is run in batch mode using the configuration information contained in the given file (for example, `IBCONF -b gpib.cfg`). See the *IBCONF Batch Mode* section later in this chapter.
`-d`	Dynamic configuration. When you exit `IBCONF`, this option causes `IBCONF` to automatically update the driver loaded in memory. See the *Exiting IBCONF* section later in this chapter.
`-f`	Disable dynamic configuration. When you exit `IBCONF`, this option causes `IBCONF` to ignore the driver loaded in memory. `IBCONF` does not try to update the loaded driver.

(continues)

Table 2-1. IBCONF Options (continued)

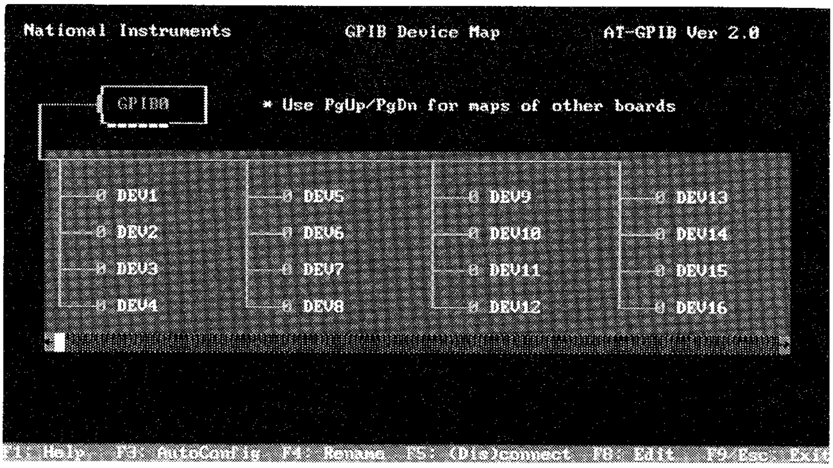
IBCONF Option	Action
`-e`	Expert mode. When you exit `IBCONF`, this option causes `IBCONF` to not to issue warning messages. See the *Exiting IBCONF* section later in this chapter.
`-m`	Monochrome mode. This option causes `IBCONF` to run in monochrome mode even though you have a color monitor.
`-vb`	Video access through BIOS. This option causes `IBCONF` to use the system BIOS routines when writing to the screen. This is slower than the default of direct screen access, but is more compatible with certain nonstandard operating environments.

Upper and Lower Levels of IBCONF

`IBCONF` operates at both an upper and a lower level. The upper level consists of the board device maps and gives a graphical picture of the GPIB system as defined in the driver. The lower level consists of screens describing the individual board and devices that make up the system.

Upper Level Device Map for Board GPIBx

Figure 2-1 shows the upper level of IBCONF.



```
National Instruments          GPIB Device Map          AT-GPIB Ver 2.0

   GPIB0        * Use PgUp/PgDn for maps of other boards

   0 DEV1      0 DEV5      0 DEV9      0 DEV13
   0 DEV2      0 DEV6      0 DEV10     0 DEV14
   0 DEV3      0 DEV7      0 DEV11     0 DEV15
   0 DEV4      0 DEV8      0 DEV12     0 DEV16

F1: Help  F3: AutoConfig  F4: Rename  F5: (Dis)connect  F8: Edit  F9/Esc: Exit
```

Figure 2-1. Upper Level of IBCONF

As shown in Figure 2-1, the upper-level screen of IBCONF displays the names of all devices controlled by the driver. It also indicates which devices, if any, are accessed through the interface or access board named GPIB*x*, where *x* is 0 or 1 for a two-board driver. You can move around the map by using the cursor control keys. If you have a Microsoft compatible mouse, you can also use it to move around the map.

The following options are available at the upper level.

Device Maps of the Boards

Help

Rename

(Dis)connect

Edit

Output GPIB Driver Configuration

- Autoconfigure
- Exit

Device Maps of the Boards

Use <PageUp> or <PageDown> to toggle between the device maps for the different GPIB interface boards. These boards are referred to as access boards. The maps show which devices are assigned to each board. By default, an equal number of devices are attached to each GPIB interface board.

Help

Use the function key F1 to access the comprehensive, online help feature of `IBCONF`. The help information describes the functions and common terms associated with the upper-level of `IBCONF`.

Rename

Use the function key F4 to rename a device. Move to the device you wish to rename by using the cursor control keys. Press the F4 key and enter the new name of the device. The device name may contain up to eight characters and uses the same rules as MS-DOS for naming files, except that suffixes (`.xxx`) are not allowed.

As specified by MS-DOS, the device name cannot use the following characters:

```
.     "     /     \     [     ]     :
|     <     >     +     =     ;     ,
```

(ASCII characters less than hex 21)

and cannot use the following reserved names:

CON NUL

Uppercase and lowercase letters are treated the same. The string `PLOTTER` is treated the same as the string `plotter`, for example. For this reason, `IBCONF` maps all lowercase letters to uppercase.

Note: You must not give GPIB device names the same names as files, directories, and/or subdirectories. If you name a GPIB device `PLTR` and your file system contains the file `PLTR.DAT` or a subdirectory `PLTR`, a conflict results. Please note that the access board names, such as `GPIB0`, cannot be altered.

The string representing a device or access board name is the first variable argument of the function `ibfind` called at the beginning of your application program. Refer to Chapter 4, *NI-488.2 Software Characteristics and Routines*, and Chapter 5, *NI-488 Software Characteristics and Functions*, of this manual for more explanations of `ibfind`.

(Dis)connect

Use the function key F5 to logically connect or disconnect a device from a board. Move the cursor to the device that you want to connect or disconnect by using the cursor control keys and press the F5 key.

Edit

Use the function key F8 or the <Enter> key to edit or examine the characteristics of a particular board or device. Move to the board or device that you want to edit using the cursor control keys and press the F8 key. This step puts you in the lower level of `IBCONF` and lists the characteristics for the particular board or device that you want to edit. To exit edit, use function key F9 or <Escape>.

Output GPIB Driver Configuration

When configuring a GPIB driver, a text version of the driver can be written to a disk file. Use the function key F2 to direct `IBCONF` to create a text file named `GPIB.TXT` in the current directory. This file contains a description of the current GPIB driver and should be used for information purposes only.

Autoconfigure

Use the function key F3 to cause `IBCONF` to perform its automatic configuration. When asked to autoconfigure a particular GPIB board, `IBCONF` interrogates all the listen addresses on the GPIB to detect listening devices. `IBCONF` then adjusts the device map for the board so that only the responding devices are connected. It also adjusts the primary and secondary address fields of the devices to match the addresses that responded as Listeners. The entire operation only takes a few seconds. You may want to rename the connected devices with names that indicate their function.

It is important that all the devices in the system be connected and powered on before running autoconfigure.

`IBCONF` disconnects devices from higher numbered interface boards to configure a lower numbered board. For this reason, if you have more than one interface board in your system, you should plan to autoconfigure all of them, beginning with board 0 and increasing. If you want to autoconfigure only one board, and leave the other one alone, you should arrange for board 1 to be the autoconfigured board, and for 0 to be the board left alone.

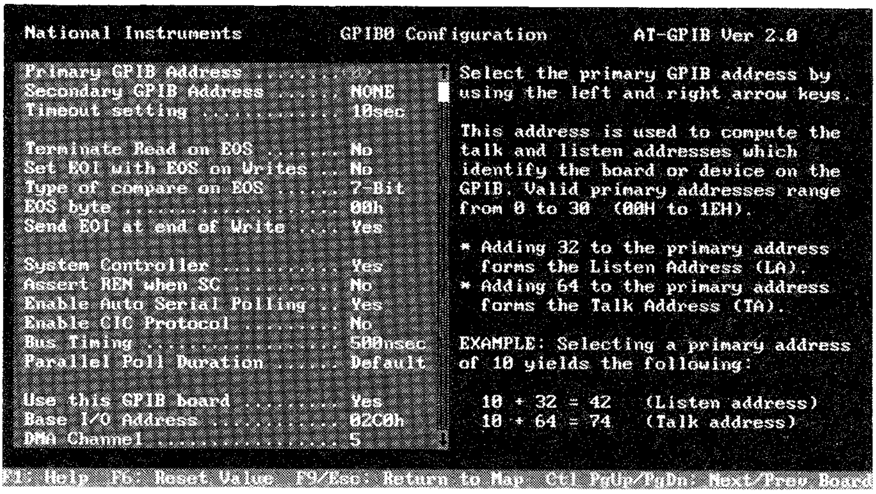
Note: Do *not* use Autoconfigure if you are using LabWindows.

Exit

Use the function key F9 or <Escape> to exit `IBCONF`. If you have made changes, `IBCONF` displays the prompt `Save changes?`. Type a `y` (yes) to save changes, `n` (no) to lose changes, or `c` (cancel) to remain in IBCONF. For more information, refer to the *Exiting IBCONF* section later in this chapter.

Lower Level Device/Board Characteristics

Figure 2-2 shows the lower level of `IBCONF`.



```
National Instruments        GPIB0 Configuration        AT-GPIB Ver 2.0

Primary GPIB Address ......... 0       Select the primary GPIB address by
Secondary GPIB Address ...... NONE     using the left and right arrow keys.
Timeout setting ............. 10sec
                                       This address is used to compute the
Terminate Read on EOS ....... No       talk and listen addresses which
Set EOI with EOS on Writes .. No       identify the board or device on the
Type of compare on EOS ...... 7-Bit    GPIB. Valid primary addresses range
EOS byte .................... 00h      from 0 to 30  (00H to 1EH).
Send EOI at end of Write .... Yes
                                       * Adding 32 to the primary address
System Controller ........... Yes        forms the Listen Address (LA).
Assert REN when SC .......... No       * Adding 64 to the primary address
Enable Auto Serial Polling .. Yes        forms the Talk Address (TA).
Enable CIC Protocol ......... No
Bus Timing .................. 500nsec  EXAMPLE: Selecting a primary address
Parallel Poll Duration ...... Default  of 10 yields the following:

Use this GPIB board ......... Yes        10 + 32 = 42   (Listen address)
Base I/O Address ............ 02C0h      10 + 64 = 74   (Talk address)
DMA Channel ................. 5

F1: Help  F6: Reset Value  F9/Esc: Return to Map  Ctl PgUp/PgDn: Next/Prev Board
```

Figure 2-2. Lower Level of IBCONF

The lower level screens of `IBCONF` display the currently defined values for characteristics of a device or board, such as addressing and timeout information, as shown in Figure 2-2. You access these screens from the upper level of `IBCONF` by selecting a board or device and pressing the function key F8 or the <Enter> key. The configuration settings selected for each device and each board are a means of customizing the communications and other options used with that board or device. The settings for devices specify the characteristics used by the access board for that device when device functions are used. The settings for boards specify the characteristics used when board functions are used.

The following functions are available at the lower level.

- Change Characteristics
- Change Board or Device
- Help

- Reset Value
- Return to Map

Change Characteristics

To change a specific characteristic of a device or a board, move the cursor to or click the mouse on that characteristic. You can also use <PageUp>, <PageDown>, <Home>, or <End> to move around the characteristics of a device or a board. When the cursor is on the characteristic, either use the left/right arrow keys to select between different options or input the option directly from the keyboard. Instructions on the right side of the screen inform you which method is appropriate for the selected characteristic.

Change Board or Device

Use <Control-PageUp> and <Control-PageDown> to change to the next or previous GPIB board or device. For example, if you are editing DEV3 and type <Control-PageUp>, you will then be editing DEV4.

Help

Use the function key F1 to access the comprehensive, online help feature of `IBCONF`. The help information describes the functions and common terms associated with the lower level of `IBCONF`.

Reset Value

Use the function key F6 to reset a characteristic option to its previous value.

Return to Map

At the lower level, the function key F9 or <Escape> returns you to the upper level device map of `IBCONF`.

Default Configurations

The NI-488.2 for MS-DOS software has factory default configurations. For example, the default device names of the 32 GPIB devices are `DEV1` through `DEV32`. You may want to change the names to more descriptive ones, such as METER for a digital multimeter.

You may use `IBCONF` to look at the current default settings in the software file.

If you do not make changes to the NI-488.2 for MS-DOS software using `IBCONF`, the default configurations of the software remain in effect.

Default Values

The following are the default values of the software.

- There are 32 devices with symbolic names `DEV1` through `DEV32`.
- There are four access boards with symbolic names `GPIB0`, `GPIB1`, `GPIB2`, and `GPIB3`. The access board names cannot be changed.
- Access board `GPIB0` is enabled. `GPIB1`, `GPIB2`, and `GPIB3` are disabled.
- The GPIB addresses of the first 16 devices are the same as the device number. For example, `DEV1` is at address 1. These devices are assigned to the access board `GPIB0`.
- The remaining 16 devices (that is, devices 17 through 32) are assigned to the access board `GPIB1`. Their GPIB addresses range from 1 through 16, respectively. For example, `DEV17` is at address 1.
- Each GPIB interface board is System Controller of its independent bus and has a GPIB address of 0.
- The END message is sent with the last byte of each data message to a device. No End-of-String (EOS) character is recognized.
- The time limit on I/O and wait function calls is set for approximately 10 sec.

- Each GPIB interface board has its own default setting for the base I/O address, interrupt setting, and DMA channel. Check the Getting Started manual that came with your interface board for these settings.

Device and Board Characteristics

The following explanations are for board and device characteristics in `IBCONF` that are common to all revisions of the NI-488.2 for MS-DOS driver. For information on characteristics specific to a given driver, check the Getting Started manual that came with your interface board. In addition, extensive help for each characteristic is displayed on the `IBCONF` screen while the cursor is positioned in a field. Most of the following characteristics apply to both devices and boards although some, as indicated, only apply to boards.

Primary GPIB Address

All devices and boards must be assigned unique primary addresses in the range from hex 00 to hex 1E (0 to 30 decimal). The driver automatically forms a listen address by adding hex 20 to the primary address. It forms the talk address by adding hex 40 to the primary address. For example, a primary address of hex 10 would have a listen address of hex 30 and a talk address of hex 50. The GPIB primary address of any device is set within that device, either with hardware switches or a software program. This address and the address listed in `IBCONF` must correspond. Refer to the device documentation for instructions about the device address. The default primary address of all GPIB boards is 0. There are no hardware switches on the GPIB interface board to select the GPIB address.

Secondary GPIB Address

Any device or board using extended addressing must be assigned a secondary address in the range from hex 60 to hex 7E (96 to 126 decimal), or the option `NONE` can be selected to disable secondary addressing. As with primary addressing, the secondary GPIB address of a device is set within that device, either with hardware switches or a software program. This address and the address listed in `IBCONF` must correspond. Refer to the device documentation for instructions about secondary addressing. Secondary addressing is disabled for all boards and devices unless changed in `IBCONF`. The default option for this characteristic is `NONE`.

Timeout Settings

The timeout value is the approximate minimum length of time that I/O functions such as `ibrd`, `ibwrt`, and `ibcmd` can take before a timeout occurs. It is also the length of time that the `ibwait` function waits for an event before returning if the TIMO bit is set in the event mask. If the SRQI bit and TIMO bit in the event mask are passed to the `ibwait` function and no SRQ is detected, the function will timeout. Refer to the *IBWAIT* function description in Chapter 3, *Understanding the NI-488.2 Software*, and Chapter 5, *NI-488 Software Characteristics and Functions*, of this manual for more information. The default option for this characteristic is `10sec`.

Serial Poll Timeouts (Device Characteristic Only)

This timeout value controls the length of time the driver waits for a serial poll response from a device. The IEEE-488 specification does not specify the length of time a Controller should wait for the response byte. The driver's default of 1 second should work for most devices. If you seem to have problems with serial polls, try using a longer timeout value.

Terminate READ on EOS

Some devices send an EOS byte signaling the last byte of a data message. A `yes` response to this field causes the GPIB board to terminate a read operation when it receives the EOS byte. The default option for this characteristic is `no`.

Set EOI with EOS on Write

A `yes` response to this field causes the GPIB board to assert the EOI line when the EOS byte is detected on a write operation. The default option for this characteristic is `no`.

Type of Compare on EOS

This field specifies the type of comparison to be made with the EOS byte. You may indicate whether all eight bits are to be compared or just the seven least significant bits (ASCII or ISO format). This field is only valid if a `yes` response was given for either the Set EOI with EOS on Write field or the Terminate Read on EOS field. The default option for this characteristic is `7-bit`.

EOS Byte

You can program some devices to terminate a read operation when a selected character is detected. A linefeed character (hex 0A) is a common EOS byte.

Note: The driver does not automatically append an EOS byte to the end of data strings on write operations. You must explicitly include this byte in your data string. The designation of the EOS byte is only for the purpose of informing the driver of its value so that I/O can terminate correctly.

The default option for this characteristic is `00H`.

Set EOI at End of Write

Some devices, as Listeners, require that the Talker terminate a data message by asserting the EOI line with the last byte. A `yes` response causes the GPIB interface board to assert the EOI line on the last data byte. The default option for this characteristic is `yes`.

System Controller (Board Characteristic Only)

This field appears on the board characteristics screen only. The System Controller in a GPIB system is the device that maintains ultimate control over the bus. There should be at most one device designated System Controller in any GPIB system. In some situations, such as a network of computers linked by the GPIB interface board, another device may be System Controller and the GPIB board should be designated as *not* System Controller. A `no` response would designate *not* System Controller and a `yes` response would designate System Controller capability. In general, the GPIB board should be designated as System Controller. The default option for this characteristic is `yes`.

Assert REN when SC (Board Characteristic Only)

A `yes` response to this field causes Remote Enable (REN) to be asserted automatically whenever the board is placed online, provided that the board has been given System Controller capability. If a `no` response is provided, an explicit call to `ibsre` is required to assert REN. The default option for this characteristic is `no`.

Enable Auto Serial Polling (Board Characteristic Only)

This option enables or disables automatic serial polls of devices when the GPIB Service Request (SRQ) line is asserted. Positive poll responses are stored following the polls and can be read with the `ibrsp` device function. Refer to Chapter 5, *NI-488 Software Characteristics and Functions*, of this manual for further information. Normally, this feature does not conflict with devices that conform to the IEEE-488 specification. If there is a conflict with a device, a `no` response for this field disables this feature. The default option for this characteristic is `no`.

Note: Turn off Auto Serial Polling if you are using the ON SRQ feature of the driver.

Enable CIC Protocol (Board Characteristic Only)

If a device-level NI-488 call is made after control has been passed to another device, enabling this protocol causes the board to assert SRQ with a Serial Poll status byte of hex 42. The current Controller must recognize that the board want to regain control. If the current Controller passes control back to the board, the device call proceeds as intended. If control is not passed within the timeout period, the error ECIC results from the call. If the CIC protocol is disabled, ECIC is returned immediately if a device call is made after control has been passed. The default option for this characteristic is `yes`.

Bus Timing (Board Characteristic Only)

This field specifies the T1 delay of the source handshake capability for the board. This delay determines the minimum amount of time, after the data is placed on the bus, that the board may assert DAV during a write or command operation. If the total length of the GPIB cable length in the system is less than 15 m, then the value of 350 nsec is appropriate.

There are other factors that may affect the choice of the T1 delay, although they are unlikely to affect you. Refer to the ANSI/IEEE488.2-1987 Standard, Section 5.2, for more information about these other factors.

The default option is `500nsec`.

Parallel Poll Duration (Board Characteristic Only)

This field specifies the length of time the driver waits when conducting a parallel poll. For a normal bus configuration (the Controller and devices on the same bus) the default duration is correct. If you are using a GPIB Bus Extender in Transparent parallel poll mode, you should increase the poll duration to 10 µsec. This allows the bus extender to operate transparently to your applications.

Enable Repeat Addressing (Device Characteristic Only)

Normally, devices are addressed each time a read or write operation is performed. If `no` is selected, read or write operations do not readdress the selected device if the same operation was just performed with that device. This saves some time when you have several GPIB operations to perform after repeat addressing.

Use This Board (Board Characteristic Only)

If you do not want the driver to try to access a board at the selected base address (because you do not have a board in the system), select `no` for this option. When this field is set to `no`, the driver does not try to access the board's hardware. It returns the error ENEB as soon as a program tries to access the board.

Base I/O Address (Board Characteristic Only)

This field specifies the I/O address of the GPIB board. It must be set to the same value as the base I/O address setting for the GPIB board itself. Setting the base I/O address level is explained in the Getting Started manual that you received with your GPIB interface board. On Micro Channel and some other systems, you can change the I/O address only by booting the reference diskette. In this case, this field is Read-only.

DMA Channel (Board Characteristic Only)

This field specifies the DMA channel to be used by the GPIB interface board. It must be set to the same value as the DMA channel (arbitration level) setting for the GPIB board itself. Setting the DMA channel level is explained in the Getting Started manual that you received with your GPIB board. On Micro Channel and some other systems, you can change the DMA channel only by booting the reference diskette. However, you can enable or disable the use of DMA with `IBCONF`.

Interrupt Level (Board Characteristic Only)

This field specifies the interrupt line used by the GPIB interface board. It must be set to the same value as the interrupt level setting for the GPIB board itself. Setting the interrupt level is explained in the Getting Started manual that you received with your GPIB board. On Micro Channel and some other systems, you can change the interrupt level only by booting the reference diskette. However, you can enable or disable the use of interrupts with `IBCONF`.

GPIB-PCII/IIA Mode Switch

The driver that is included with the GPIB-PCII and GPIB-PCIIA interface board kits is the same for each kit and can run with both boards. With the GPIB-PCII/IIA Mode Switch field, you can select the type of board that is installed in your system. It is possible to have both a GPIB-PCII and a GPIB-PCIIA interface board installed in your system at the same time.

Exiting IBCONF

After you have made all your changes, you can exit `IBCONF` by pressing function key F9 or <Escape>. The program first displays the prompt `Save changes?` before exiting. Typing a `y` response causes the changes to be written to the file on disk.

Before exiting, the program checks for situations that can cause problems, such as the following:

- Is there a GPIB addressing conflict between a device and its access board?

- Are GPIB boards not present in the host machine at the specified address?
- Are timeouts disabled on a device or board?

If any of these situations are found, you are notified and given the option of re-entering or exiting `IBCONF`. To disable automatic checking when starting `IBCONF`, enter the following command:

```
ibconf -e
```

You can also have `IBCONF` modify the driver that is loaded in memory if the loaded version is compatible with the version on the disk. If the loaded version is compatible and you did not start `IBCONF` with the `-f` or `-d` option, a prompt asks you if you want to modify the loaded driver. If you enter `y`, the currently loaded driver is modified with your parameter selections. If you enter `n`, the loaded driver is left alone. This prompt does not appear if `IBCONF` cannot find a loaded driver. If you have started `IBCONF` with the `-d` option, the prompt for the configuration of the loaded version is not displayed and the loaded version is configured automatically, if it is compatible. By using the `-f` option, the configuration of the loaded version is skipped.

If you have made and saved changes, and you do *not* modify the resident driver, you must restart the computer for your changes to take effect. If you have changed a board interrupt level or changed a device name to `LPT1`, `LPT2`, `LPT3`, `COM1`, or `COM2`, `IBCONF` does not configure the loaded version.

IBCONF Batch Mode

The `IBCONF` batch mode offers an alternate method for modifying the configurations of the NI-488.2 driver.

Note: Batch mode can also modify the loaded driver. Refer to the *Exiting IBCONF* section earlier in this chapter.

In batch mode, the configuration information is contained in a configuration file that you create. To use the configuration file that you have created, type in the following command:

```
ibconf -b filename          <Enter>
```

where `filename` is the name of the configuration file, for example, `sample.cfg`. There must be at least one space between `-b` and `filename`.

The following example is a sample configuration file along with a step-by-step explanation of each entry.

Example: `sample.cfg` sample configuration file

1. Change the name of the first device to plotter.
2. Disconnect the second device.
3. Connect the third device to board 1.
4. Change the configuration of board 0 (`gpib0`).
    - Primary address to 2 (`pad 2`)
    - Timeout setting to 30 seconds (`tmo 30sec`)
    - EOS byte to hex 1E (`eos 0x1E`)
    - System Controller capability to NO (`sc no`)
    - Type of compare on EOS to 8-bit (`bin 8-bit`)
    - GPIB bus timing to 350 nsec (`tmng 350nsec`)

```
find device1 name plotter
find device2 disconnect
find device3 connect board1
find board0 pad 2 tmo 30sec eos 0x1E sc no
            bin 8-bit tmng 350nsec
```

The configuration file is a free-form text file consisting of pairs of items. Each item must be separated by at least one space or new line character–this is the only format restriction. The first item of a pair is a mnemonic that represents a board or device characteristic, or a board/device map configuration function (for example, `rename`, `connect`, `disconnect`) to be configured. The second item of a pair is the value to which the first item (mnemonic) should be set.

Before configuring a board or device, that board or device must be found by the `find name#` pair, where `name` is `board` or `device` and `#` is the index of the GPIB board or device you are configuring (see the `sample.cfg` example).

When `IBCONF` is run in batch mode, it checks the syntax of the first item in the pair, the value range of the second item, and reports any errors as it finds them. If a value range error is found, the correct range of values is displayed. If any errors are found, the driver will not be configured.

Table 2-2 contains a list of all the valid pairs of items. Any value that can be entered as a number can be entered in decimal or hexadecimal. Hexadecimal numbers must be preceded with `0x` (for example, decimal 64 would be written as `0x40`).

Table 2-2. Batch Mode Command Pairs

First Item		**Second Item**	
Mnemonic	**Explanation**	**Value**	**Notes**
`find`	Find board or device	`board#` or `device#`	
`pad`	Primary GPIB address	number	1
`sad`	Secondary GPIB address	number	4
`tmo`	Timeout settings	mnemonic	2
`xeos`	Set EOI with EOS on write	`yes` or `no`	
`bin`	Type of compare on EOS	`7-bit` or `8-bit`	
`eot`	Set EOI with last byte of write	`yes` or `no`	

(continues)

Table 2-2. Batch Mode Command Pairs (continued)

First Item		Second Item	
Mnemonic	**Explanation**	**Value**	**Notes**
`sc`	System Controller (board only)	`yes` or `no`	
`sre`	Assert REN when SC (board only)	`yes` or `no`	
`spoll`	Enable auto serial polling (board only)	`yes` or `no`	
`tmng`	Timing (board only)	`2µsec`, `500nsec`, `350nsec`	
`cic_prot`	CIC protocol (board only)	`yes` or `no`	
`int`	Interrupt setting (board only)	number	4
`port`	Base I/O address (board only)	number	1
`dma`	DMA channel (board only)	number	4
`raddr`	Repeat addressing (device only)	`yes` or `no`	
`name`	Rename a device (device only)	device name	3
`connect`	Connect a device to a board (device only)	`board#`	
`disconnect`	Disconnect a device from a board (device only)	no value	
`type`	Switch the current board to PC2 or PC2A mode	`PCII` or `PCIIA`	5
`pplength`	Parallel poll duration (board only)	mnemonic	6
`useboard`	Use this interface board (board only)	`yes` or `no`	
`spolltmo`	Serial Poll Timeout (device only)	mnemonic	2

Notes for Table 2-2

1. Refer to the function descriptions in Chapter 5, *NI-488 Software Characteristics and Functions*, for appropriate values.

2. Timeout mnemonics are as follows:

```
NONE,     1msec,    1sec,
10usec,   3msec,    3sec,
30usec,   10msec,   10sec,
100usec,  30msec,   30sec,
300usec,  100msec,  100sec
          300msec,
```

3. The name must be a valid DOS name up to eight characters long, without a suffix.

4. The value must be either a valid secondary address, parallel poll enable, DMA channel, or interrupt level; or the word *none* to disable. Refer to the function descriptions in Chapter 5, *NI-488 Software Characteristics and Functions*, for appropriate values.

5. This option is only valid on the GPIB-PCII/IIA driver.

6. Use any of the mnemonics (except `NONE`) listed in Note 2 to set the parallel poll duration. The mnemonic `DEFAULT` should be used to select the default parallel poll duration.

Dynamic Reconfiguration of Board and Device Characteristics

You can call functions during the execution of an application program to dynamically change the configured values. These functions are shown in Table 2-3.

Table 2-3. Functions That Alter Default Characteristics

Characteristic	Dynamically Changed by
Primary GPIB address	`ibpad`
Secondary GPIB address	`ibsad`
End-of-String (EOS) byte	`ibeos`
7- or 8-bit compare on EOS	`ibeos`

(continues)

Table 2-3. Functions That Alter Default Characteristics (continued)

Characteristic	Dynamically Changed by
Terminate a Read on EOS	`ibeos`
Set EOI with last byte of Write	`ibeot`
Change board assignment	`ibbna`
Enable or disable DMA	`ibdma`
Change or disable time limit	`ibtmo`
Request/release system control	`ibrsc`

The function `ibconfig` is used to dynamically set these and almost all other parameters at run-time. See the description of this function in Chapter 5, *NI-488 Software Characteristics and Functions*.

Using Your NI-488.2 Software

The NI-488.2 software consists of a high-speed driver and several utilities to help in developing and debugging an application program. The NI-488.2 driver can be accessed in the following three ways: directly with the NI-488 functions, with the NI-488.2 routines, or through a lower-performance character I/O driver that uses HP-style commands.

NI-488 Functions and NI-488.2 Routines

The NI-488.2 software is a subroutine-structured device driver. The NI-488.2 driver is faster than other available device drivers, easily handles buffered DMA transfers and uses a structured, hierarchical programming style familiar to users of modern programming languages. The NI-488 functions and NI-488.2 routines are described in Chapters 3, 4, and 5 of this manual. An NI-488.2 or NI-488 language interface is required to link application programs to the driver. The distribution diskette has the language interfaces for BASICA, Microsoft QuickBASIC, Microsoft BASIC, and Microsoft C.

The following is a QuickBASIC example of a high-level NI-488 function that writes an array of bytes to a device:

```
data$ = "F0R2S2"
CALL ibwrt (scope%, data$)
```

Universal Language Interface

The Universal Language Interface (ULI) described in the *Universal Language Interface Using HP-Style Calls* manual (National Instruments part number 320135-90) uses a character I/O driver to access the NI-488.2 software. You can gain access to the ULI from the standard I/O commands of virtually any language or spreadsheet by passing HP-style command strings. Like any character I/O driver, the ULI performs much more slowly than the NI-488 functions and the NI-488.2 routines. Although the driver can be called from many different programming languages, the ULI is best suited for BASIC language users familiar with HP commands. A language interface is not required when using the ULI.

The following is a QuickBASIC example of a ULI function that writes a command to a device:

```
PRINT #1, "OUTPUT 1;F0R2S2"
```

Interactive Control Program (IBIC)

A good way to begin learning your GPIB system is to use the Interface Bus Interactive Control, `IBIC`, program described in Chapter 6, *IBIC*. With `IBIC`, you can program your instruments interactively from the keyboard rather than from an application program. Using `IBIC` helps you quickly understand how the instruments and the NI-488.2 software work. It also immediately returns the same status information that is returned as global variables in an application program.

While running `IBIC`, study the descriptions of each function given in Chapter 6 to fully understand the purpose of each function or use the online help available in `IBIC`.

Applications Monitor

The Applications Monitor is a DOS memory-resident program that is used for automatic error detection and debugging of command sequences. The Applications Monitor is described in Chapter 7, *Applications Monitor*. You can automatically halt program execution after any GPIB command or only on error conditions so that you can examine the function arguments, data buffers, return values, and global variables. The Applications Monitor also stores up to 255 of the preceding GPIB commands so that you can trace sequences of commands. The Applications Monitor replaces the explicit error checking code of an application program.

The Application Program

When you decide to write your application program, be sure to refer to the appropriate language reference manuals (or Chapters 4 and 5 of this manual) for the proper syntax of the functions. If there is a problem with the program, install the Applications Monitor. Also, use `IBIC` to test the sequence of commands your application program uses.

Chapter 3
Understanding the NI-488.2 Software

This chapter introduces you to the NI-488.2 routines and NI-488 functions, describes features common to both, and contains general programming information you need to know before starting your application program. It also contains specific programming information for each language.

- The NI-488.2 routines directly adhere to the Controller sequences and protocols defined in the ANSI/IEEE488.2-1987 standard. They accept a single device address or a list of device addresses as an input parameter so that functions can address multiple instruments easily. These routines give you all the advantages of 488.2.
- The NI-488 functions have existed for many years and are a *de facto* industry standard for MS-DOS GPIB applications. They have both high-level, device functions and low-level, board functions.

This chapter also discusses programming issues such as global variables, error codes, read and write termination, and BASIC/QuickBASIC/BASIC/C programming preparations that are common to both the NI-488.2 routines and NI-488 functions.

Introduction to the NI-488.2 Routines

A new set of NI-488.2 routines have been added to the NI-488.2 for MS-DOS software to take advantage of the IEEE-488.2-1987 standard. The NI-488.2 routines are described in Chapter 4, *NI-488.2 Software Characteristics and Routines* . These routines are completely compatible with the Controller sequences and protocols defined in the IEEE-488.2-1987 standard.

IEEE-488.2 is the standard upon which the new generation of test systems will be built because it enhances system compatibility and configurability by defining data formats, status reporting, Controller capabilities and commands, and a general command set to which all 488.2 instruments must adhere. 488.2 is also the basis of the Standard Commands for Programmable Instrumentation (SCPI), so all SCPI instruments are 488.2 compatible. The NI-488.2 routines address these system programming benefits of 488.2.

The syntax of the NI-488.2 routines resembles the naming conventions used in the standard. These routines let you take full advantage of 488.2, especially when a complete 488.2 system of Controllers and instruments are used. There are routines that find all of the Listeners on the bus, configure the attached instruments, find a device requesting service, determine the state of the SRQ line, wait for SRQ to be asserted, and address multiple devices. If your application plans call for IEEE-488.2, it is best to use the NI-488.2 routines.

Some programming implementations, such as configuring timeout values or monitoring all of the bus management lines, are not specifically described in 488.2. For these requirements, the traditional NI-488 board functions can be used along with the NI-488.2 routines. The necessary NI-488 board functions are described in the *Relationship of NI-488.2 Routines to NI-488 Calls* section at the beginning of Chapter 4.

Introduction to the NI-488 Functions

The NI-488 functions consist of high-level (or device) functions that hide much of the GPIB management operations and low-level (or board) functions that offer you complete control over the GPIB. Typically, only a few high-level functions are needed for most application programs. These functions are described in Chapter 5.

Device Functions

Device functions are high-level functions that are easy to learn and use. These functions free you from having to know the GPIB protocol or bus management details involved. They automatically execute sequences of commands that handle bus management operations required to perform activities such as reading from and writing to devices or polling them for status. Device functions access a specific device and take care of the addressing and bus management protocol for that device. A descriptor of the accessed device is one of the function's arguments.

Board Functions

In contrast, board functions are low-level functions that perform rudimentary GPIB operations. They are necessary because high-level functions may not always meet the requirements of applications. In such cases, low-level functions offer the flexibility to solve your application needs.

Board functions access the GPIB interface board directly and require you to do the addressing and bus management protocol for the bus. A descriptor of the accessed board is one of the function's arguments.

More About Device and Board Functions

You may find it helpful to compare how a high-level device function can be replaced by several low-level board functions. Conducting a serial poll is a good example. In the discussion of the `ibrsp` function, the following BASIC example of the device function is used:

```
CALL ibrsp (pltr%,status%)
```

This is equivalent to the following sequence using the board functions just described:

```
cmd$ = "?!" + chr$(&H18) + "G"
CALL ibcmd (gpib0%,cmd$)
status$ = space$(1)
CALL ibrd (gpib0%,status$)
cmd$ = chr$(&H19) + "_"
CALL ibcmd (gpib0%,cmd$)
```

The first `ibcmd` function is used to send the string of ASCII commands assigned in the first program line. These are Unlisten (`?`), listen address of the board (`!`) with primary address 1, Serial Poll Enable (`chr$(&H18)`), and talk address of the plotter with primary address 7 (`G`). Now that the plotter is addressed to send its status byte and the board is addressed to receive it, the `ibrd` function is called to read the byte and store it in the variable `status`. The final `ibcmd` function completes the poll by sending the command string consisting of two messages: Serial Poll Disable (`chr$(&H19)`) and Untalk (_).

You can see that a high-level device function is easier to use. However, if an application requires a more complex serial poll routine than the one just described, such as one that polls several devices in succession and has other servicing operations at the same time, you can use board functions to create such a routine.

Opening Boards and Devices

The first step when using an NI-488 function is to obtain the unit descriptor for all boards and devices that you are using. The unit descriptor `ud` is the general reference to the board or device descriptor returned by the `ibfind` function or the `ibdev` function. A unit descriptor of a device as the first argument in a function specifies a device function. A unit descriptor of a board as the first argument in a function specifies a board function. Some NI-488 functions may be both a board function and a device function.

IBFIND (board or devname, dev)

`ibfind` returns a unit descriptor associated with the name of boards or devices and must be called before any other NI-488 functions. When the software is installed, a description of each device is placed in an internal reference table accessible by the software. The `ibfind` function locates a board or device using the symbolic names defined in the software such as `gpib0`, `dev1`, or `scope`. To find out the names of these symbols, you can run `IBCONF`.

IBDEV (boardindex,pad,sad,tmo,eot, eos,ud)

An alternative to `ibfind` is the `ibdev` function. `ibdev` is a function that returns a unit descriptor for a device. It is used when the symbolic name of the device is unimportant and the values of the configurable parameters are specified.

Programming Features Common to NI-488.2 Routines and NI-488 Functions

This section describes programming characteristics that apply when using either the NI-488.2 routines or the NI-488 functions.

Multiboard Driver

The driver can control or manipulate more than one interface board. Figure 3-1 shows a multiboard GPIB system with board `gpib0` connected to two devices (an oscilloscope and a digital voltmeter) and with board `gpib1` connected to two other devices (a printer and a plotter). This type of driver is commonly called a multiboard driver.

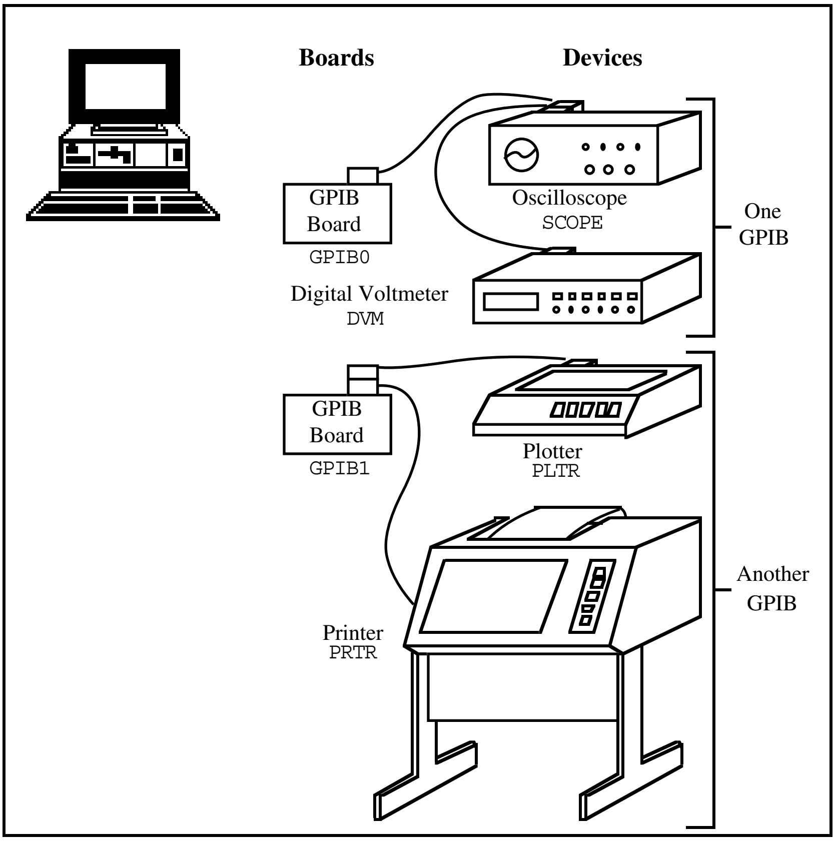


Figure 3-1. Multiboard GPIB System

Learning NI-488.2 and Your Instruments

The best way to learn the NI-488.2 routines and NI-488 functions and the commands of your instruments is interactively through your keyboard. The Interface Bus Interactive Control (`IBIC`) program lets you input both NI-488 functions and NI-488.2 routines from the keyboard. You can easily control instruments and receive status and error information without writing an application program. `IBIC` is described with step-by-step instructions in Chapter 6.

General Programming Information

The following facilities or operations are common to all programming options and languages:

- Status Word (`ibsta`)
- Error Codes (`iberr`)
- Count Variables (`ibcnt`, `ibcntl`)
- Read and Write Termination

You should understand these topics thoroughly to take full advantage of the NI-488.2 for MS-DOS software capabilities.

The next several paragraphs explain the status word (`ibsta`), the error variable (`iberr`), and the count variables (`ibcnt`, `ibcntl`). These variables are updated after each function to reflect the status of the device or board just accessed.

Status Word – ibsta

All functions return a status word containing information about the state of the GPIB and the GPIB interface board. You can test for the conditions reported in the status word to make decisions about continued processing. The status word is returned in the variable `ibsta`. In addition, in all languages except BASICA, calls can be made as *functions* (as opposed to *subroutines*) and the status word is returned as the integer value of the function.

The status word contains 16 bits, all of which are meaningful. A bit value of one (1) indicates that the corresponding condition is in effect. A bit value of zero (0) indicates that the condition is not in effect.

Table 3-1 lists the conditions and the bit position to test for each condition. Some bits are set only on device functions (dev); some bits are set only on board functions (brd); and some bits are set on either type (dev, brd). The NI-488.2 routines are considered board functions.

Table 3-1. Status Word (`ibsta`) Layout

Mnemonics	**Bit Pos.**	**Hex Value**	**Function Type**	**Description**
ERR	15	8000	dev, brd	GPIB error
TIMO	14	4000	dev, brd	Time limit exceeded
END	13	2000	dev, brd	END or EOS detected
SRQI	12	1000	brd	SRQ interrupt received
RQS	11	800	dev	Device requesting service
EVENT	10	400	brd	A DTAS or DCAS event has occurred.
SPOLL	9	200	brd	The board has been serial polled by the Controller.
CMPL	8	100	dev, brd	I/O completed
LOK	7	80	brd	Lockout State
REM	6	40	brd	Remote State
CIC	5	20	brd	Controller-In-Charge
ATN	4	10	brd	Attention is asserted
TACS	3	8	brd	Talker

(continues)

Table 3-1. Status Word (`ibsta`) Layout (continued)

Mnemonics	Bit Pos.	Hex Value	Function Type	Description
LACS	2	4	brd	Listener
DTAS	1	2	brd	Device Trigger State
DCAS	0	1	brd	Device Clear State

The declaration file for BASICA/QuickBASIC/BASIC/C defines the mnemonic for each bit in the status bytes ibsta and iberr. For example, if you are programming in QuickBASIC, the following two calls are equivalent:

- `IF IBSTA% AND TACS THEN PRINT "TALK ADDRESS"`
- `IF IBSTA% AND &H0008 THEN PRINT "TALK ADDRESS"`

A description of each status word and its condition follows.

ERR (dev, brd) ERR is set in the status word following any call that results in an error; the particular error may be determined by examining the `iberr` variable. ERR is cleared following any call that does not result in an error.

Note: It is recommended that you check for an error condition after each call either explicitly or with the Applications Monitor. An unnoticed error occurring early in your application program may not become apparent until a later instruction. At that time, the error can be more difficult to locate.

TIMO (dev, brd) TIMO indicates whether a timeout has occurred. TIMO is set in the status word following an `ibwait` if the TIMO bit of the `ibwait` mask parameter is also set and if the wait has exceeded the time limit value. TIMO is also set following any synchronous I/O functions (for example, `ibrd`, `ibwrt`, and `ibcmd`) if a timeout occurs during a call. TIMO is cleared in the status word in all other circumstances.

END (dev, brd)

END indicates either that the END message has been received from the EOI line or that the software is configured to terminate a read function on an EOS byte and that an EOS byte has been received following a read function. While the GPIB board is performing a shadow handshake as a result of the `ibgts` function, any other function may return a status word with the END bit set if the END or EOS message occurred before or during that call. END is cleared in the status word when any I/O operation is initiated.

Some applications may need to know the exact I/O read termination mode of a read operation: EOI by itself, the EOS character by itself, or EOI plus the EOS character. You can use the `ibconfig` function to enable a mode in which the END bit is set only when EOI is received. If the I/O operation completes because of the EOS character by itself, END will not be set. The application should check the last byte of the received buffer to see if the last byte is the EOS character.

SRQI (brd)

SRQI specifies that some device is requesting service. SRQI is set in the status word whenever the GPIB board is CIC, the GPIB SRQ line is asserted, and the automatic serial poll capability is disabled. SRQI is cleared whenever the GPIB board ceases to be the CIC, or the GPIB SRQ line is unasserted.

RQS (dev)

RQS appears only in the status word of a device function and indicates that the device is requesting service. RQS is set in the status word whenever the hex 40 bit is asserted in the serial poll status byte of the device. The serial poll that obtains the status byte may be the result of an `ibrsp`, or the poll may be done automatically by the software if automatic serial polling is enabled. RQS is cleared when an `ibrsp` reads the serial poll status byte that caused the RQS. An `ibwait` on RQS should only be done on devices that respond to serial polls.

EVENT (brd)

This bit can be used in Talker/Listener applications (applications in which the GPIB interface is not the Controller) to monitor the GPIB device clear and group execute trigger commands. A Talker/Listener application may need to record the order of device clear

	and trigger events. The usual DCAS and DTAS bits of `ibsta` are not sufficient because they cannot indicate which event happened first. The EVENT bit is disabled by default. If you want to use this bit, you must use the `ibconfig` function to enable this bit. When this bit is enabled, the DCAS and DTAS bits are disabled. When an event occurs, this bit is set and any I/O in progress is aborted. The application then can call the `ibevent` function to determine which event occurred.
SPOLL (brd)	This bit can be used in Talker/Listener applications to determine when the GPIB board has been serial polled by the Controller. The SPOLL bit is disabled by default. Use the `ibconfig` function to enable it. When this bit is enabled, it sets after the board has been polled, but only if the board was requesting service at the time. See the description of `ibrsv` in Chapter 5 for more information. SPOLL is cleared from the status word on any call immediately after an `ibwait` call if the SPOLL bit was set in the wait mask, or immediately following a call to `ibrsv`.
CMPL (dev, brd)	CMPL indicates the condition of outstanding I/O operations. It is set in the status word whenever I/O is complete. CMPL is cleared while I/O is in progress.
LOK (brd)	LOK indicates whether the board is in a lockout state. While LOK is set, the `EnableLocal` routine or `ibloc` function is inoperative for that board. LOK is set whenever the GPIB board detects the Local Lockout (LLO) message has been sent either by the GPIB board or by another Controller. LOK is cleared when the Remote Enable (REN) GPIB line becomes unasserted by the System Controller.
REM (brd)	REM indicates whether or not the board is in the remote state. REM is set whenever the Remote Enable (REN) GPIB line is asserted and the GPIB board detects that its listen address has been sent either by the GPIB board or by another Controller. REM is cleared when REN becomes unasserted, when the GPIB board as a Listener detects that the Go to Local (GTL) command has been

	sent either by the GPIB board or by another Controller, or when the `ibloc` function is called while the LOK bit is cleared in the status word.
CIC (brd)	CIC indicates whether the GPIB board is the Controller-In-Charge. CIC is set when the `SendIFC` routine or `ibsic` function is executed while the GPIB board is System Controller or when another Controller passes control to the GPIB board. CIC is cleared whenever the GPIB board detects Interface Clear (IFC) from the System Controller, or when the GPIB board passes control to another device.
ATN (brd)	ATN indicates the state of the GPIB Attention (ATN) line. ATN is set whenever the GPIB ATN line is asserted and cleared when the ATN line is unasserted.
TACS (brd)	TACS indicates whether the GPIB board has been addressed as a Talker. TACS is set whenever the GPIB board detects its talk address (and secondary address, if enabled) has been sent either by the GPIB board itself or by another Controller. TACS is cleared whenever the GPIB board detects the Untalk (UNT) command, its own listen address, a talk address other than its own talk address, or Interface Clear (IFC).
LACS (brd)	LACS indicates whether the GPIB board has been addressed as a Listener. LACS is set whenever the GPIB board detects its listen address (and secondary address, if enabled) has been sent either by the GPIB board itself or by another Controller. LACS is also set whenever the GPIB board shadow handshakes as a result of the `ibgts` function. LACS is cleared whenever the GPIB board detects the Unlisten (UNL) command, its own talk address, Interface Clear (IFC), or `ibgts` is called without shadow handshake.
DTAS (brd)	DTAS indicates whether the GPIB board has detected a device trigger command. DTAS is set whenever the GPIB board, as a Listener, detects the Group Execute Trigger (GET) command has been sent by another Controller. DTAS is cleared in the status word on any call immediately following an `ibwait` call if the DTAS bit is set in the `ibwait` mask parameter.

DCAS (brd)	DCAS indicates whether the GPIB board has detected a device clear command. DCAS is set whenever the GPIB board detects the Device Clear (DCL) command has been sent by another Controller, or whenever the GPIB board as a Listener detects the Selected Device Clear (SDC) command has been sent by another Controller. DCAS is cleared in the status word on any call immediately following an `ibwait` call if the DCAS bit was set in the `ibwait` mask parameter, or on any call immediately following a read or write.

In the event that a function call returns an ENEB or EDVR error, all status word bits except the ERR bit are cleared, because these error codes indicate that it is not possible to obtain the status of the GPIB board.

Error Variable – iberr

If the ERR bit is set in the status word, a GPIB error has occurred–that is, if the previous GPIB call returned with an `ibsta` value in which the ERR bit is set, the following interpretations of `iberr` apply. The error code is returned in the variable `iberr`. Table 3-2 lists these error codes.

Table 3-2. GPIB Error Codes

Suggested Mnemonic	**Decimal Value**	**Explanation**
EDVR	0	DOS error
ECIC	1	Function requires GPIB board to be CIC
ENOL	2	Write handshake error (e.g., no Listener)
EADR	3	GPIB board not addressed correctly
EARG	4	Invalid argument to function call
ESAC	5	GPIB board not System Controller as required
EABO	6	I/O operation aborted (timeout)
ENEB	7	Nonexistent GPIB board
EDMA	8	Windows 3 only. Virtual DMA device error
EOIP	10	Asynchronous I/O in progress
ECAP	11	No capability for operation
EFSO	12	File system error
EBUS	14	GPIB bus error
ESTB	15	Serial Poll status byte queue overflow
ESRQ	16	SRQ stuck in ON position
ETAB	20	Table problem

A description of each error and some conditions under which each error may occur follows:

EDVR (0) EDVR is returned when the device or board name passed in an `ibfind` call is not configured in the software. In this case, the variable `ibcnt` contains the DOS error code 2, "Device not found." The remedy is to replace the argument to `ibfind` with a valid board or device name or to reconfigure the software using the `IBCONF` utility to recognize the name. EDVR is also returned when an invalid unit descriptor is passed to any function call. In this case, the variable `ibcnt` contains the DOS error code 6, "Invalid handle." The remedy is to be sure that `ibfind` has been called and that it completed successfully. Also note that following a call to `ibonl` with a second argument of 0, which places `brd` *offline*, an `ibfind` is required before any subsequent calls to or using that device or board.

EDVR is also returned when the software (`GPIB.COM`) is not installed. Check the `CONFIG.SYS` file in the root directory and make sure it contains the line:

```
DEVICE=dir\GPIB.COM
```

where `dir` is the directory that contains the file `GPIB.COM` For example, `dir` might be `C:\AT-GPIB`.

ECIC (1) ECIC is returned when one of the following board functions or routines is made while the board is not CIC:

- Any of the NI-488.2 routines
- Any board functions that put command bytes on the GPIB bus: `ibcmd`, `ibcmda`, `ibln`, `ibrpp`
- `ibcac`, `ibgts`
- Any device functions that affect the GPIB

In cases when the GPIB board should always be the CIC, be sure to call `SendIFC` or `ibsic` to send Interface Clear before attempting any of these calls, and to avoid sending the command byte TCT (hex 09, Take Control). In multiple CIC situations, always be certain that the CIC bit appears in the status word `ibsta` before attempting these calls. If it is not, it is possible to perform an `ibwait` (CIC) call to delay further processing until control is passed to the board.

ENOL (2) ENOL usually occurs when a write operation is attempted with no Listeners addressed. For a device write, this error indicates that the GPIB address configured for that device in the software does not match the GPIB address of any device connected to the bus, that the GPIB cable is not connected to the device, or that the device is not powered on. This situation can be corrected by either attaching the appropriate device to the GPIB, modifying the address of an already attached device, calling `ibpad` (and `ibsad` if necessary) to match the configured address to the device switch settings, or using the `IBCONF` configuration utility to reassign the proper GPIB address to the device in the software.

For board functions, an `ibcmd` is generally necessary to address devices before a board `ibwrt` function is executed. Be sure that the proper listen address is in the `ibcmd` argument string and that no Unlisten (hex 3F) command follows it.

ENOL may occur in situations in which the GPIB board is not the CIC and the Controller asserts ATN before the write call in progress has ended. To clear this error, reduce the write byte count to that which is expected by the Controller, or to resolve the situation on the end of the Controller.

EADR (3) EADR occurs when the GPIB board is CIC and is not addressing itself before read and write functions. This error is unlikely to occur on a device function. For a board function, be sure to send the appropriate Talk or Listen address using `SendCmds`, `SendSetup`, or `ReceiveSetup` before attempting `SendDataBytes` or `RcvRespMsg`; or `ibcmd` before attempting `ibwrt` or `ibrd`.

EADR is also returned by the function `ibgts` when the shadow-handshake feature is requested and the GPIB ATN line is already unasserted. In this case, the shadow handshake is not possible and the error is returned to notify you of that fact. Avoid calling `ibgts` except immediately after an `ibcmd` call. (`ibcmd` causes ATN to be asserted.)

EARG (4)	EARG results when an invalid argument is passed to a function call. The following are some examples: `ibtmo` called with a value not in the range 0 through 17. `ibeos` called with meaningless bits set in the high byte of the second parameter. `ibpad` or `ibsad` called with invalid addresses. `ibppc` called with invalid parallel poll configurations. A board function made with a valid device descriptor, or a device-function made with a valid board descriptor. **Note**: EDVR is returned if the descriptor is invalid.
ESAC (5)	ESAC results when `SendIFC`, `ibsic`, `EnableRemote`, or `ibsre` is called when the GPIB board does not have System Controller capability. The remedy is to give the GPIB board that capability by calling `ibrsc` or by using `IBCONF` to configure that capability into the software.
EABO (6)	EABO indicates that I/O has been canceled, usually due to a timeout condition. Other causes are `ibstop` being called or the Device Clear message being received from the CIC. To remedy a timeout error, if I/O is actually progressing but times out anyway, lengthen the timeout period with `ibtmo`. More frequently, however, the I/O is stuck (the Listener is not continuing to handshake or the Talker has stopped talking), or the byte count in the call which timed out was more than the other device was expecting. Be sure that both parties to the transfer understand what byte count is expected; or, if possible, have the Talker use the END message to assist in early termination.
ENEB (7)	ENEB occurs when there is no GPIB board at the I/O address specified in the configuration program. This happens when the board is not physically plugged into the system, the I/O address specified during configuration does not match the actual board setting, or there is a conflict in the system with the base I/O address. If there is a mismatch between the actual board setting and the value specified at configuration

time, either reconfigure the software or change the board switches to match the configured value.

EDMA (8) EDMA can only occur in Windows 3 386 Enhanced mode when Windows uses a virtual DMA device with a problem. There are two reasons a problem can exist with the virtual DMA device:

- If you run version 3.0 of Windows with DMA enabled for the GPIB board, and you do not use the replacement virtual DMA device `nivdmad.386` provided by National Instruments, this error occurs. To use DMA in this situation, refer to the manual *Using Your NI-488.2 Software with Microsoft Windows* (National Instruments part number 320319-01) for details on using the replacement DMA device.
- If you run version 3.1 of Windows or if you run version 3.0 of Windows with `nivdmad.386` and EDMA occurs, the error means that the virtual DMA device is in an unknown state and cannot necessarily perform a DMA transfer. Although this case is not likely to occur, the virtual DMA device can become unstable when another device driver in the system does not properly use the virtual DMA device. The only solution to this problem is to restart Windows. This returns the virtual DMA device to a known state.

EOIP (10) EOIP occurs when an asynchronous I/O operation has not finished before some other call is made. During asynchronous I/O, only `ibstop`, `ibwait`, and `ibonl` calls are allowed. If any other call is attempted, EOIP is returned.

The three asynchronous I/O calls (`ibcmda`, `ibrda`, and `ibwrta`) are designed to allow an application to perform other functions (non-GPIB functions) while processing the I/O. Once the asynchronous I/O call has been initiated, further GPIB calls involving the device or access board are not allowed until the I/O has completed and the GPIB driver and the application have been resynchronized.

Resynchronization can be accomplished by using one of the following three functions:

Note: Resynchronization is only successful if the `ibsta` returned contains CMPL.

- `ibwait` - The driver and application are synchronized.
- `ibstop` - The asynchronous I/O is canceled, and the driver and application are synchronized.
- `ibonl` - The asynchronous I/O is canceled, the interface has been reset, and the driver and application are synchronized.

The only other GPIB call that is allowed during asynchronous I/O is the `ibwait` function (mask is arbitrary). Any other GPIB call involving the device or access board returns the EOIP error.

ECAP (11) ECAP results when a particular capability has been disabled in the software and a call is made that attempts to make use of that capability.

EFSO (12) EFSO results when an `ibrdf` or `ibwrtf` call encounters a problem performing a file operation. Specifically, this error indicates that the function is unable to open, create, seek, write, or close the file being accessed. The specific DOS error code for this condition is contained in `ibcnt`.

EBUS (14) EBUS results when certain GPIB bus errors occur during device functions. All device functions send command bytes to perform addressing and other bus management. Devices are expected to accept these command bytes within the time limit specified by the configuration program or by `ibtmo`. EBUS results if a timeout occurred during the sending of these command bytes. Under normal operating circumstances, the remedy would be to find out which GPIB device is accepting commands abnormally slow and fix the problem with that device. In situations in which you want slow handshaking of the commands, lengthen the board time limit either with the configuration program or the `ibtmo` function.

ESTB (15) ESTB occurs only during the `ibrsp` function. ESTB indicates that one or more serial poll status bytes received from automatic serial polls have been discarded. The bytes were discarded because of a lack of storage space. Several

	older status bytes are available; however, the oldest is being returned by the `ibrsp` call. If your application cannot tolerate missing even one status byte, the remedy is to disable Automatic Serial Polling using `IBCONF`.
ESRQ (16)	ESRQ occurs only during the `WaitSRQ` routine or `ibwait` function. ESRQ indicates that a wait for RQS is not possible because the GPIB SRQ line is stuck on. The usual reason for this situation is that a device, unknown to the software, is asserting SRQ. Because the software does not know of this device, it can never be serial polled and SRQ can never unassert. Another reason for this error is that a GPIB bus tester or similar equipment was forcing the SRQ line to be asserted or that there is a cable problem involving the SRQ line. Although the occurrence of ESRQ signals a definite GPIB problem, it does not affect GPIB operations, except that you cannot depend on the RQS bit while the condition lasts.
ETAB (20)	ETAB occurs only during the `FindLstn` and `FindRQS` routines and the `ibevent` function. ETAB indicates that there was some problem with a table used by these functions. In the case of `FindLstn`, this is not an error condition but simply an advisory message which means that the given table did not have enough room to hold all the addresses of the Listeners found. In the case of `FindRQS`, it means that none of the devices in the given table were requesting service. In the case of `ibevent`, the event queue has overflowed and the most recent events are lost. Call `ibevent` more often to keep the queue empty.

Count Variables – ibcnt and ibcntl

The count variables are updated after each read, write, or command function with the number of bytes actually transferred by the operation. They are also updated by many of the NI-488.2 routines. `ibcnt` is an integer value (16 bits wide) and `ibcntl` is a long integer value (32 bits wide).

Read and Write Termination

The IEEE-488 specification defines two methods of identifying the last byte of device-dependent (data) messages. The two methods permit a Talker to send data messages of any length without the Listener(s) knowing in advance the number of bytes in the transmission. The two methods are as follows:

- END message. In this method, the Talker asserts the EOI (End Or Identify) signal simultaneously with transmission of the last data byte. By design, the Listener stops reading when it detects a data message accompanied by EOI, regardless of the value of the byte.

- End-of-String (EOS) character. In this method, the Talker uses a special character at the end of its data string. With IEEE-488.2, this EOS character is explicitly designated as a new line (NL, ASCII 10, hex 0A) character. By prior arrangement, the Listener stops receiving data when it detects that character. Either a 7-bit ASCII character or a full 8-bit binary byte can be used.

The NI-488.2 software terminates reads and writes according to the value of the termination parameter. For reads, `STOPend` terminates the read operation on the END message. If the termination parameter is a character, the read operation terminates when that character is received. For writes, the termination parameter may be either `NLend`, `DABend`, or `NULLend`. `NLend` sends NL with END after the last data byte has been sent; `DABend` sends END with the last data byte to be sent; and `NULLend` simply sends the data bytes with no END message.

BASICA/QuickBASIC/BASIC/C Programming Information

The following discussion contains information for programming the driver functions in BASICA, QuickBASIC, BASIC, and C.

Note: The term *BASICA*, as used in this section, refers to Advanced IBM Interpretive BASIC for the IBM Personal Computer. The term *QuickBASIC* refers to Microsoft QuickBASIC. The term *BASIC* refers to Microsoft Professional BASIC. The term *C* refers to Microsoft C.

This programming information can be used with all versions of BASICA, QuickBASIC, and C unless otherwise specified.

BASICA Files

The NI-488 software distribution diskette contains the following files which are relevant to programming in BASICA:

- `DECL.BAS` is a file containing initialization code.
- `BIB.M` is the BASICA language interface which gives your application program access to the driver.
- `DBSAMP.BAS` is a sample program using device calls.
- `BBSAMP.BAS` is a sample program using board calls.
- `BSAMP488.BAS` is a sample program using 488.2 routines.

QuickBASIC Files

The NI-488 software distribution diskette contains the following files which are relevant to programming in QuickBASIC, versions 4.0 and later.

- `QBDECL.BAS` is the QuickBASIC initialization code.
- `QBIB.OBJ` is the QuickBASIC language interface.
- `DQBSAMP.BAS` is a sample program using device functions.
- `BQBSAMP.BAS` is a sample program using board functions.
- `QSAMP488.BAS` is a sample program using 488.2 routines.

BASIC Files

The NI-488 software distribution diskette contains the following files which are relevant to programming in BASIC, version 7.0.

- `MBDECL.BAS` is the BASIC initialization code.
- `MBIB.OBJ` is the BASIC language interface.
- `DMBSAMP.BAS` is a sample program using device functions.

- `BMBSAMP.BAS` is a sample program using board functions.
- `MSAMP488.BAS` is a sample program using 488.2 routines.

C Language Files

The NI-488 software distribution diskette contains the following files which are relevant to programming in C, version 5.0 and later:

- `DECL.H` is a file containing useful variable and constant declarations.
- `MCIB.OBJ` is the interface which gives your application program access to the driver.
- `DCSAMP.C` is a sample program using device calls.
- `BCSAMP.C` is a sample program using board calls.
- `CSAMP488.C` is a sample program using 488.2 functions.

Programming Preparations for BASICA

A BASICA language code block must be executed before the main body of your application program. The file `DECL.BAS` already contains this code block.

Place the file `DECL.BAS` at the beginning of your application program with appropriate adjustments to line numbers using the BASICA MERGE command. `DECL.BAS` contains code that loads the BASICA language interface file `BIB.M` into memory. `BIB.M` must exist in the directory currently in use.

You may change function and variable names defined in the code block if they conflict with names used in your application program. The substitution must be done consistently and carefully throughout. The instructions in this chapter assume that no name substitution is necessary and that the recommended names are used.

The GPIB status, error, and count information is returned in the variables `ibsta%`, `iberr%`, and `ibcnt%`, respectively, as described in the appropriate sections at the beginning of this chapter.

For BASICA, all function arguments are either integer or string variables. The values of these must be assigned before the function call is made. The `CLEAR` statement in line 1 of `DECL.BAS` contains a constant that represents the amount of memory that can be used after loading the BASICA language interface `BIB.M`. For the great majority of users, this constant is correct. If you have a system with an extremely small amount of memory, this constant may be incorrect. If, after BASICA is invoked, it reports that there are fewer than 60,000 bytes free, you need to adjust the constant to a smaller value.

Programming Preparations for QuickBASIC

`QBDECL.BAS` must be included at the beginning of your application program. `QBDECL.BAS` defines all the NI-488 subroutines and functions that can be used in your program.

The GPIB status, error, and count information is returned in the variables `ibsta%`, `iberr%`, and `ibcnt%` and `ibcntl%`, respectively, as described in the appropriate sections at the beginning of this chapter.

The file `QBIB.OBJ` is the QuickBASIC language interface to the NI-488.2 for MS-DOS software. Link compiled QuickBASIC application programs with `QBIB.OBJ`. You can link from the DOS command prompt by using the QuickBASIC linker, or you can set up the QuickBASIC environment to run your programs from within the environment.

You must create an object module library if you are going to make an `EXE` file in the QuickBASIC environment. To create an object module library called `QBIB.LIB`, enter the following command:

```
LIB QBIB.LIB + QBIB.OBJ;
```

To set up the QuickBASIC environment, you must first create a QuickLibrary with the language interface. For example, to create a QuickLibrary called `QBIB.QLB`, enter the following command:

```
LINK /Q QBIB.OBJ,,,BQLB45.LIB;
```

You may want to create a QuickLibrary using a Library supplied by another manufacturer in conjunction with the National Instruments language interface. The following is an example of creating a QuickLibrary with the Microsoft Library `QB.LIB` and `QBIB.OBJ`.

```
   LINK /Q QB.LIB + QBIB.OBJ,QBIB,,BQLB45.LIB;
```

For QuickBASIC 4.0, substitute `BQLB40.LIB` for `BQLB45.LIB` in the above commands. Then, to run QuickBASIC using this QuickLibrary, enter:

```
   QB /L QBIB.QLB
```

To run your program from MS-DOS, follow the instructions in the Microsoft *QuickBASIC* manuals that pertain to compiling and linking programs.

Programming Preparations for BASIC

`MBDECL.BAS` must be included at the beginning of your application program. `MBDECL.BAS` defines all the NI-488 subroutines and functions that can be used in your program.

The GPIB status, error, and count information is returned in the variables `ibsta%`, `iberr%`, and `ibcnt%` and `ibcntl%`, respectively, as described in the appropriate sections at the beginning of this chapter.

The file `MBIB.OBJ` is the BASIC language interface to the NI-488.2 for MS-DOS software. Link compiled BASIC application programs with `MBIB.OBJ`. You can link from the DOS command prompt by using the BASIC linker, or you can set up the QBX environment to run your programs from within the environment.

You must create an object module library if you are going to make an `EXE` file in the QBX environment. To create an object module library called `MBIB.LIB`, enter the following command:

```
   LIB MBIB.LIB + MBIB.OBJ;
```

To set up the QBX environment, you must first create a Library with the language interface. For example, to create a Library called `MBIB.LIB`, enter the following command:

```
LINK /Q MBIB.OBJ,,,QBXQLB.LIB;
```

You may want to create a Library using a Library supplied by another manufacturer in conjunction with the National Instruments language interface. The following is an example of creating a Library with the Microsoft Library `QBX.LIB` and `MBIB.OBJ`:

```
LINK /Q QBX.LIB + MBIB.OBJ,MBIB,,QBXQLB.LIB;
```

To run BASIC using this QuickLibrary, enter the following command:

```
QBX /L MBIB.QLB
```

To run your program from MS-DOS, follow the instructions in the Microsoft *Professional BASIC* manuals that pertain to compiling and linking programs.

Programming Preparations for C

Include the following C statement at the beginning of your application program:

```
#include "decl.h"
```

The file `MCIB.OBJ`, is the C language interface for the NI-488.2 for MS-DOS software. The compiled C application program written is linked with the `MCIB.OBJ` to produce an executable file.

"ON SRQ" Capability

The NI-488.2 for MS-DOS driver can be interrupted whenever the GPIB SRQ signal is asserted. When this interrupt occurs, the program may go to a user-specified service routine that determines the source of the SRQ and takes the appropriate action. The GPIB SRQ line activity is monitored by the light pen of the computer. Once the service routine has completed, control is passed back to the next statement in the executing statement. Example code for BASICA/QuickBASIC/BASIC and C are shown in the following paragraphs.

Note: For ON PEN to function properly, you must disable the configuration parameter *Enable Auto Serial Polling*–that is, select NO). Also, many mouse drivers use the same light pen interrupt; therefore, you may need to disable your mouse.

BASICA/QuickBASIC/BASIC "ON SRQ" Capability

BASICA/QuickBASIC/BASIC programs may be interrupted on the occurrence of the GPIB SRQ signal. When the interrupt occurs, the program may go to a user-specified service routine.

The NI-488.2 for MS-DOS driver uses the BASICA/QuickBASIC/BASIC "ON PEN" statement to intercept SRQ interrupts and make them available to user programs. For more complete information regarding the operation of "ON PEN," refer to the IBM *BASIC Reference Manual*, the Microsoft *QuickBASIC* manuals, and the Microsoft *Professional BASIC* manuals under the *"ON PEN" Statement.* All the light pen information in the *"ON PEN" Statement* section applies to the GPIB SRQ signal.

The code structure for implementing the "ON SRQ" capability in BASIC and QuickBASIC is as follows:

```
    10 ON PEN GOSUB XXXX
    20 REM Enable ON SRQ capability.
    30 PEN ON
            .
            .
            .
XXXX  REM ON SRQ Service Routine
                .
                .
                .
            RETURN
```

The special "ON PEN" feature of BASICA and QuickBASIC is not usually a feature of non-BASIC languages.

C "ON SRQ" Capability

C programs may be interrupted on the occurrence of the GPIB SRQ signal upon completion of any GPIB function. When the interrupt occurs, the program may go to a user-specified service routine. A special function, `ibsrq`, is used to make the NI-488.2 for MS-DOS driver check for the occurrence of an SRQ after any GPIB function has completed. `ibsrq` is a board function that is passed the address of the user specified service routine that executes upon the assertion of an SRQ.

The code structure for implementing the "ON SRQ" capability in C is as follows:

```
void far srqroutine()
  {
          .
          .
          .
  }
main() {

    int board;

    board = 0;

    ibsrq(srqroutine);

          .
          .
          .

}
```

The next two chapters describe in detail the NI-488.2 routines and NI-488 functions, respectively. Complete descriptions of each routine and function are given and example code and programs are shown.

Chapter 4
NI-488.2 Software Characteristics and Routines

This chapter contains a discussion of the important characteristics of the NI-488.2 routines available in the MS-DOS driver that are common to all programming languages. It also contains a detailed description of each NI-488.2 routines with example programs. The description are listed alphabetically for easy reference.

Overview

The IEEE-488.2 1987 specification explains in greater detail than the earlier IEEE-488.1 specification the exact ways in which the GPIB is to be managed by the Controller, the standard messages that compliant devices should understand, the mechanisms by which device errors and other status information are reported, and various protocols aimed at discovering which compliant devices are connected to the bus and configuring these devices.

To be fully compliant with the IEEE-488.2 protocol, the latest revisions of many National Instruments GPIB interface boards are now fully compatible with the more stringent requirements of the IEEE-488.2 specification. In addition, routines have been added to the software. By using these routines, you can have a programming interface that closely resembles the descriptions found in the IEEE-488.2 specification document, and that strictly adheres to the command and data sequences found there.

Using National Instruments NI-488.2 routines together with compliant 488.2-compatible devices can result in greater predictability of instrument behavior and programming correctness, and an increased similarity in the way that instruments of different manufacturers are programmed.

General Programming Information

The NI-488.2 routines use the Controller protocols and procedures described in the IEEE-488.2 specification. The calling syntax of the routines is intended to closely resemble the implementations suggested in that specification document.

The NI-488.2 set of routines consists of the following routines, whose functionality can be broken down into the following groups:

- Simple Device I/O

```
- Send
- Receive
```

- Multiple Device I/O

```
- SendList
```

- Simple Device Control

```
- Trigger
- DevClear
- ReadStatusByte
- PPoll
- PPollConfig
- PPollUnconfig
- PassControl
```

- Multiple Device Control

```
- TriggerList
- DevClearList
- EnableRemote
- EnableLocal
- FindRQS
- AllSpoll
```

- Bus Management

```
- ResetSys
- SendIFC
- FindLstn
- TestSRQ
- WaitSRQ
- TestSys
- SendLLO
- SetRWLS
- GenerateREQF
- GenerateREQT
- GotoMultAddr
```

- Low-Level I/O

    - `SendCmds`
    - `SendDataBytes`
    - `SendSetup`
    - `RcvRespMsg`
    - `ReceiveSetup`

The Simple Device I/O routines can read and write to individual GPIB devices.

The Multiple Device I/O routines can write the same message to multiple Listeners with a single message transmission.

The Simple Device Control routines direct various bus management instructions to individual devices.

The Multiple Device Control routines direct bus management instructions to multiple devices in the same message.

The Bus Management routines cause system-wide functions to be performed, or provide system-wide status.

The Low-Level I/O routines are used to break down a higher-level routine into more detailed instructions due to unusual situations.

All routines take, as their first parameter, a board number selecting a GPIB interface board installed in the computer. For the majority of cases, there will be only one GPIB interface board installed, and its board number will be 0. Therefore, in the typical case, a 0 is the first argument of all NI-488.2 routines.

Routines that operate on single devices have, as their second parameter, an integer that indicates the GPIB address of the device. In the typical case of a device that uses only primary GPIB addressing, this involves passing a simple integer in the range 0 to 30, corresponding to the primary GPIB address of the device. In the more unusual case of a device with both a primary and a secondary address, the two addresses are packed into an integer with the primary address in the lower byte and the secondary address in the higher byte. In QuickBASIC and BASIC, such integers would resemble, for example, &H6103, which indicates a device whose primary address is 3 and whose secondary address is hex 61.

Note: In C, a macro has been defined in the header file `DECL.H`, `MakeAddr(p, s)`, which can be used to pack the primary and secondary addresses into the correct form.

Routines that operate on multiple devices have, as their second parameter, an integer array containing the addresses in question. The individual addresses are formed in the same way as described in the previous paragraph for the single-device routines, except that they are placed in consecutive elements of an integer array, followed by a special value, `NOADDR`, to mark the end of the addresses.

I/O routines contain a buffer argument and, in some languages, a count argument. If present, the count argument may be specified as either an integer or a long integer (if allowed by the language.)

Some routines require other parameters to fulfill particular specialized needs of the routine.

Relationship of NI-488.2 Routines to NI-488 Calls

The NI-488.2 routines have a complete set of Controller procedures and protocols as defined in IEEE-488.2. There are cases, however, where more detailed control of the GPIB is required, in ways that are outside the scope of the IEEE-488.2 standard. These situations include the following:

- Communicating with non-compliant (non-488.2) devices
- Altering various low-level board configurations
- Managing the bus in non-typical ways

The original National Instruments NI-488 board functions are compatible with, and can be interspersed within, sequences of NI-488.2 routines. For example, a call to `ibtmo` can be issued within sequences of NI-488.2 routines to alter the timeout value; a call to `iblines` can be issued to monitor the state of any given GPIB line, and so on. To make these calls from within a sequence of NI-488.2 routines, the usual call to `ibfind` is not required; merely substitute the board number as the first parameter of the NI-488 board function. Thus, all calls in the sequence, both NI-488.2 and NI-488, will have the same board number (usually 0) as the first parameter. Using these calls as needed within an NI-488.2 program ensures that non-standard or unusual situations or devices can be dealt with easily.

Timeouts

Most of the NI-488.2 routines, particularly those involving the transfer of command sequences or data messages, are regulated by the same timeout mechanism that regulates the NI-488 calls. A default timeout period of 10 seconds is preconfigured in the driver; thus, all I/O must complete within that period to avoid a timeout error. In addition, the `WaitSRQ` routine waits for this period before returning with a “no SRQ” indication. The default timeout value can be changed with the `IBCONF` utility. In addition, you can use the NI-488 board function call `ibtmo` to programmatically alter the timeout period. Refer to the description of the `ibtmo` function in Chapter 5, *NI-488 Software Characteristics and Functions*, for more information.

Regardless of the I/O and Wait timeout period, a much shorter timeout is enforced for responses to serial polls. This shorter timeout period takes effect whenever a serial poll is conducted. Because devices normally respond quickly to polls, there is no need to wait for the relatively lengthy I/O timeout period for a non-responsive device.

BASICA/QuickBASIC/BASIC/C NI-488.2 Routines

Tables 4-1, 4-2, and 4-3 list the call syntax for each NI-488.2 routine and a brief description of these routines for BASICA, QuickBASIC/BASIC, and C, respectively.

Table 4-1. BASICA NI-488.2 Routines

Call Syntax	Description
`AllSpoll (board%,addresslist%(0), resultlist%(0))`	Serial poll all devices
`DevClear (board%,address%)`	Clear a single device
`DevClearList (board%, addresslist%(0))`	Clear multiple devices
`EnableLocal (board%, addresslist%(0))`	Enable operations from the front of a device

(continues)

Table 4-1. BASICA NI-488.2 Routines (continued)

Call Syntax	**Description**
`EnableRemote (board%, addresslist%(0))`	Enable remote GPIB programming of devices
`FindLstn (board%,addresslist%(0), resultlist%(0),limit%)`	Find all Listeners
`FindRQS (board%,addresslist%(0), result%)`	Determine which device is requesting service
`PassControl (board%,address%)`	Pass control to another device with Controller capability
`PPoll (board%,result%)`	Perform a parallel poll
`PPollConfig (board%,address%, dataline%,sense%)`	Configure a device for parallel polls
`PPollUnconfig (board%, addresslist%(0))`	Unconfigure devices for parallel polls
`RcvRespMsg (board%,data$, termination%)`	Read data bytes from already addressed device
`ReadStatusByte (board%,address%, result%)`	Serial poll a single device to get its status byte
`Receive (board%,address%,data$, termination%)`	Read data bytes from a GPIB device
`ReceiveSetup (board%,address%)`	Prepare a particular device to send data bytes and prepare the GPIB board to read them
`ResetSys (board%,addresslist%(0))`	Initialize a GPIB system on three levels
`Send (board%,address%,data$, eotmode%)`	Send data bytes to a single GPIB device
`SendCmds (board%,commands$)`	Send GPIB command bytes
`SendDataBytes (board%,data$, eotmode%)`	Send data bytes to already addressed devices
`SendIFC (board%)`	Clear the GPIB interface functions with IFC

(continues)

Table 4-1. BASICA NI-488.2 Routines (continued)

Call Syntax	Description
`SendList (board%,addresslist%(0), data$,eotmode%)`	Send data bytes to multiple GPIB devices
`SendLLO (board%)`	Send the local lockout message to all devices
`SendSetUp (board%,addresslist%(0))`	Prepare particular devices to receive data bytes
`SetRWLS (board%,addresslist%)`	Place particular devices in the Remote with Lockout state
`TestSRQ (board%,result%)`	Determine the current state of the SRQ line
`TestSys (board%,addresslist%, resultlist%(0))`	Cause devices to conduct self-tests
`Trigger (board%,address%)`	Trigger a single device
`Triggerlist (board%, addresslist%(0))`	Trigger multiple devices
`WaitSRQ (board%,result%)`	Wait until a device asserts Service Request

Table 4-2. QuickBASIC/BASIC NI-488.2 Routines

Call Syntax	Description
`AllSpoll (board%,addresslist%(), resultlist%())`	Serial poll all devices
`DevClear (board%,address%)`	Clear a single device
`DevClearList (board%, addresslist%())`	Clear multiple devices
`EnableLocal (board%,addresslist%())`	Enable operations from the front of a device
`EnableRemote (board%, addresslist%())`	Enable remote GPIB programming of devices

(continues)

Table 4-2. QuickBASIC/BASIC NI-488.2 Routines (continued)

Call Syntax	**Description**
`FindLstn (board%,addresslist%(), resultlist%(),limit%)`	Find all Listeners
`FindRQS (board%,addresslist%(), result%)`	Determine which device is requesting service
`PassControl (board%,address%)`	Pass control to another device with Controller capability
`PPoll (board%,result%)`	Perform a parallel poll
`PPollConfig (board%,address%, dataline%,sense%)`	Configure a device for parallel polls
`PPollUnconfig (board%, addresslist%())`	Unconfigure devices for parallel polls
`RcvRespMsg (board%,data$, termination%)`	Read data bytes from already addressed device
`ReadStatusByte (board%,address%, result%)`	Serial poll a single device to get its status byte
`Receive (board%,address%,data$, termination%)`	Read data bytes from a GPIB device
`ReceiveSetup (board%,address%)`	Prepare a particular device to send data bytes and prepare the GPIB board to read them
`ResetSys (board%,addresslist%())`	Initialize a GPIB system on three levels
`Send (board%,address%,data$, eotmode%)`	Send data bytes to a single GPIB device
`SendCmds (board%,commands$)`	Send GPIB command bytes
`SendDataBytes (board%,data$, eotmode%)`	Send data bytes to already addressed devices
`SendIFC (board%)`	Clear the GPIB interface functions with IFC
`SendList (board%,addresslist%(), data$,eotmode%)`	Send data bytes to multiple GPIB devices

(continues)

Table 4-2. QuickBASIC/BASIC NI-488.2 Routines (continued)

Call Syntax	Description
`SendLLO (board%)`	Send the local lockout message to all devices
`SendSetUp (board%,addresslist%())`	Prepare particular devices to receive data bytes
`SetRWLS (board%,addresslist%)`	Place particular devices in the Remote with Lockout state
`TestSRQ (board%,result%)`	Determine the current state of the SRQ line
`TestSys (board%,addresslist%, resultlist%())`	Cause devices to conduct self-tests
`Trigger (board%,address%)`	Trigger a single device
`Triggerlist (board%,addresslist%())`	Trigger multiple devices
`WaitSRQ (board%,result%)`	Wait until a device asserts Service Request

Table 4-3. C NI-488.2 Routines

Call Syntax	Description
`AllSpoll (board,addresslist, resultlist)`	Serial poll all devices
`DevClear (board,address)`	Clear a single device
`DevClearList (board,addresslist)`	Clear multiple devices
`EnableLocal (board,addresslist)`	Enable operations from the front of a device
`EnableRemote (board,addresslist)`	Enable remote GPIB programming of devices
`FindLstn (board,addresslist, resultlist,limit)`	Find all Listeners

(continues)

Table 4-3. C NI-488.2 Routines (continued)

Call Syntax	Description
`FindRQS (board,addresslist,result)`	Determine which device is requesting service
`GenerateREQF (board, addr)`	Cancel service request
`GenerateREQT (board, addr)`	Request service
`GotoMultAddr (board, type, addrfunc, spollfunc)`	Enable multiple primary or secondary address support
`PassControl (board,address)`	Pass control to another device with Controller capability
`PPoll (board,result)`	Perform a parallel poll
`PPollConfig (board,address, dataline,sense)`	Configure a device for parallel polls
`PPollUnconfig (board,addresslist)`	Unconfigure devices for parallel polls
`RcvRespMsg (board,data,termination)`	Read data bytes from already addressed device
`ReadStatusByte (board,address, result)`	Serial poll a single device to get its status byte
`Receive (board,address,count, termination)`	Read data bytes from a GPIB device
`ReceiveSetup (board,address)`	Prepare a particular device to send data bytes and prepare the GPIB board to read them
`ResetSys (board,addresslist)`	Initialize a GPIB system on three levels
`Send (board,address,data,eotmode)`	Send data bytes to a single GPIB device
`SendCmds (board,commands,count)`	Send GPIB command bytes
`SendDataBytes (board,data,count eotmode)`	Send data bytes to already addressed devices
`SendIFC (board)`	Clear the GPIB interface functions with IFC

(continues)

Table 4-3. C NI-488.2 Routines (continued)

Call Syntax	Description
`SendList (board,addresslist,data, count,eotmode)`	Send data bytes to multiple GPIB devices
`SendLLO (board)`	Send the local lockout message to all devices
`SendSetUp (board,addresslist)`	Prepare particular devices to receive data bytes
`SetRWLS (board,addresslist)`	Place particular devices in the Remote with Lockout state
`TestSRQ (board,result)`	Determine the current state of the SRQ line
`TestSys (board,addresslist, resultlist)`	Cause devices to conduct self-tests
`Trigger (board,address)`	Trigger a single device
`Triggerlist (board,addresslist)`	Trigger multiple devices
`WaitSRQ (board,result)`	Wait until a device asserts Service Request

NI-488.2 Routine Descriptions

The remainder of this chapter contains a detailed description of each NI-488.2 routine with examples. The descriptions are listed alphabetically for easy reference.

AllSpoll AllSpoll

Purpose: Serial Poll all devices.

Format:

BASICA

```
CALL AllSpoll (board%, addresslist% (0), resultlist%
                 (0))
```

QuickBASIC/BASIC

```
CALL AllSpoll (board%, addresslist% (), resultlist% ())
```

C

```
void AllSpoll (short board, unsigned short addresslist [],
                 unsigned short resultlist [])
```

`board` specifies a board number. The GPIB devices whose addresses are contained in the address array are serial polled, and the responses are stored in the corresponding elements of the `resultlist` array. The parameter `addresslist` is an array of address integers of any size, terminated by the value `NOADDR`.

If any of the specified devices times out instead of responding to the poll, then the error code EABO is returned in `iberr`, and `ibcnt` contains the index of the timed-out device.

Although the `AllSpoll` routine is general enough to serial poll any number of GPIB devices, the `ReadStatusByte` routine should be used in the case of polling exactly one GPIB device.

AllSpoll (continued) AllSpoll

Example:

Serial poll two devices connected to board `0` whose GPIB addresses are 8 and 9.

BASICA

```
70  DIM addresslist%(3)
71  DIM resultlist%(2)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = NOADDR
100 CALL AllSpoll (board%, addresslist% (0),
resultlist%(0))
```

QuickBASIC/BASIC

```
70  DIM addresslist%(3)
71  DIM resultlist%(2)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = NOADDR
100 CALL AllSpoll (board%, addresslist% (), resultlist%())
```

C

```
unsigned short addresslist[3] = {8, 9, NOADDR};
unsigned short resultlist[2];
AllSpoll (0, addresslist, resultlist);
```

DevClear DevClear

Purpose: Clear a single device.

Format:

BASICA/QuickBASIC/BASIC

```
CALL DevClear (board%, address%)
```

C

void `DevClear` (short `board`, unsigned short `address`)

`board` specifies a board number. The GPIB Selected Device Clear (SDC) message is sent to the device at the given address. The parameter `address` contains in its low byte the primary GPIB address of the device to be cleared. The high byte should be 0 if the device has no secondary address. Otherwise, it should contain the desired secondary address. If `address` contains the constant value `NOADDR`, the Universal Device Clear message is sent to all devices on the GPIB.

The `DevClear` routine is used to clear either exactly one GPIB device, or all GPIB devices. To send a single message that clears several particular GPIB devices, use the `DevClearList` routine.

Example:

Clear a digital voltmeter connected to board `0` whose primary GPIB address is 9 and whose secondary GPIB address is 97.

BASICA/QuickBASIC/BASIC

```
80  board% = 0
90  address% = 9 + 256*97
100  CALL DevClear (board%, address%)
```

DevClear (continued) DevClear

C

```
DevClear (0, MakeAddr (9, 97));

/* In C, a macro has been defined in the header
 * file DECL.H, MakeAddr(p, s), which can be
 * used to pack the primary and secondary
 * addresses into the correct form.
 */
```

DevClearList DevClearList

Purpose: Clear multiple devices.

Format:

BASICA

```
CALL DevClearList (board%, addresslist% (0))
```

QuickBASIC/BASIC

```
CALL DevClearList (board%, addresslist% ())
```

C

```
void DevClearList (short board, unsigned short addresslist [])
```

`board` specifies a board number. The GPIB devices whose addresses are contained in the address array are cleared. The parameter `addresslist` is an array for any size of address integers, terminated by the value `NOADDR`.

Although the `DevClearList` routine is general enough to clear any number of GPIB devices, the `DevClear` routine should be used in the common case of clearing exactly one GPIB device.

If the array contains only the value `NOADDR` or if in C, a `NULL` value is passed, the Universal Device Clear message is sent.

DevClearList (continued) DevClearList

Example:

Clear two devices connected to board `0` whose GPIB addresses are 8 and 9.

BASICA

```
70  DIM addresslist%(3)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = NOADDR
100  CALL DevClearList (board%, addresslist% (0))
```

QuickBASIC/BASIC

```
DIM addresslist%(3)
addresslist% (0) = 8
addresslist% (1) = 9
addresslist% (2) = NOADDR
CALL DevClearList (0, addresslist% ())
```

C

```
unsigned short addresslist[3] = {8, 9, NOADDR};
DevClearList (0, addresslist);
```

EnableLocal EnableLocal

Purpose: Enable operations from the front panel of a device.

Format:

BASICA

```
CALL EnableLocal (board%, addresslist% (0))
```

QuickBASIC/BASIC

```
CALL EnableLocal (board%, addresslist% ())
```

C

```
void EnableLocal (short board, unsigned short addresslist [])
```

`board` specifies a board number. The GPIB devices whose addresses are contained in the `addresslist` array are placed in local mode by addressing the devices as Listeners and sending the GPIB Go To Local command. The parameter `addresslist` is an array for any size of address integers, terminated by the value `NOADDR`.

If the array contains only the value `NOADDR`, or if in C, a `NULL` value is passed for `addresslist`, Remote Enable (REN) becomes unasserted, immediately placing all GPIB devices in local mode.

EnableLocal (continued) EnableLocal

Example:

Place the devices at GPIB addresses 8 and 9 in local mode.

BASICA

```
70  DIM addresslist%(3)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = NOADDR
100  CALL EnableLocal (board%, addresslist% (0))
```

QuickBASIC/BASIC

```
DIM addresslist%(3)
addresslist% (0) = 8
addresslist% (1) = 9
addresslist% (2) = NOADDR
CALL EnableLocal (0, addresslist% ())
```

C

```
unsigned short addresslist[3] = {8, 9, NOADDR};
EnableLocal (0, addresslist);
```

EnableRemote EnableRemote

Purpose: Enable remote GPIB programming of devices.

Format:

BASICA

```
CALL EnableRemote (board%, addresslist% (0))
```

QuickBASIC/BASIC

```
CALL EnableRemote (board%, addresslist% ())
```

C

```
void EnableRemote (short board, unsigned short addresslist [])
```

`board` specifies a board number. The GPIB devices whose addresses are contained in the `addresslist` array are placed in remote mode by asserting Remote Enable (REN) and addressing the devices as Listeners. The parameter `addresslist` is an array for any size of address integers, terminated by the value `NOADDR`.

If the array contains only the value `NOADDR`, or if in C, a `NULL` value is passed for `addresslist`, no addressing is performed, and Remote Enable (REN) becomes asserted.

EnableRemote (continued) EnableRemote

Example:

Place the devices at GPIB addresses 8 and 9 in remote mode.

BASICA/QuickBASIC/BASIC

```
70  DIM addresslist%(3)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = NOADDR
100  CALL EnableRemote (board%, addresslist% (0))
```

QuickBASIC/BASIC

```
DIM addresslist%(3)
addresslist% (0) = 8
addresslist% (1) = 9
addresslist% (2) = NOADDR
CALL EnableRemote (0, addresslist% ())
```

C

```
unsigned short addresslist[3] = {8, 9, NOADDR};
EnableRemote (0, addresslist);
```

FindLstn FindLstn

Purpose: Find all Listeners.

Format:

BASICA

```
CALL FindLstn (board%, addresslist% (0), resultlist% (0),
               limit%)
```

QuickBASIC/BASIC

```
CALL FindLstn (board%, addresslist% (), resultlist% (), limit%)
```

C

```
void FindLstn      (short board, unsigned short addresslist [],
              unsigned short resultlist [], unsigned short limit)
```

`board` specifies a board number. `addresslist` contains a list of primary GPIB addresses, terminated by the value `NOADDR`. These addresses are tested in turn for the presence of a listening device. If found, the addresses are entered into the `resultlist`. If no listening device is detected at a particular primary address, all the secondary addresses associated with that primary address are tested, and detected Listeners are entered into `resultlist`. The `limit` argument specifies how many entries should be placed into the `resultlist` array. If more Listeners are present on the bus, the list is truncated after `limit` entries have been detected, and the error ETAB is reported in `iberr`. The variable `ibcnt` contains the number of addresses placed into `resultlist`.

Because for any given primary address there may be multiple secondary addresses that respond as Listeners, the `resultlist` array should, in general, be larger than the `addresslist` array. In any event, the `resultlist` (with `limit` being the maximum possible results) array must be large enough to accommodate all expected listening devices because no check is made for overflow of the array.

Because most GPIB devices have the ability to listen, this routine is normally used to detect the presence of devices at particular addresses. Once detected, they usually can be interrogated by identification messages to determine what devices they are.

FindLstn (continued) FindLstn

Example:

Determine which one of the devices at addresses 8, 9, and 10 are present on the GPIB.

BASICA

```
70  DIM addresslist%(4)
'        Because there are three primary GPIB
'        addresses, in the worst case 93
'        separate GPIB devices could be detected
'        at all the secondary addresses.  In
'        this example, we are assuming that we
'        know that there are at most 5 devices
'        connected to the GPIB.
75  DIM resultlist%(5)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = 10
93  addresslist% (3) = NOADDR
94  limit% = 5
100  CALL FindLstn (board%, addresslist% (0), resultlist (0),
limit%)
```

Following this call, `resultlist% ()` might contain the following values:

```
resultlist% (0)       9
resultlist% (1)       10 + 96*256
resultlist% (2)       10 + 99*256
```

These results indicate that three GPIB devices were detected. One was found at address 9 with no secondary address, no GPIB devices were detected at primary address 8, and, at address 10, two devices with secondary addresses were found. Because only primary GPIB addresses 8, 9 and 10 were tested, it is possible that more GPIB devices are connected at other addresses.

FindLstn (continued) FindLstn

QuickBASIC/BASIC

```
DIM addresslist%(4)
DIM resultlist%(5)
addresslist% (0) = 8
addresslist% (1) = 9
addresslist% (2) = 10
addresslist% (3) = NOADDR
limit% = 5
CALL FindLstn (0, addresslist% (), resultlist% (), limit%)
```

C

```
unsigned short addresslist[4] = {8, 9, 10, NOADDR};
unsigned short resultlist[5];
FindLstn (0, addresslist, resultlist, 5);
```

FindRQS FindRQS

Purpose: Determine which device is requesting service.

Format:

BASICA

```
CALL FindRQS (board%, addresslist% (0), result%)
```

QuickBASIC/BASIC

```
CALL FindRQS (board%, addresslist% (), result%)
```

C

```
void FindRQS      (short board,  unsigned short addresslist [],
              unsigned short *result)
```

`board` specifies a board number. `addresslist` contains a list of primary GPIB addresses, terminated by the value `NOADDR`. Starting from the beginning of the `addresslist`, the indicated devices are serial polled until one is found which is asserting SRQ. The status byte for this device is returned in the variable `result`. In addition, the index of the device in the `addresslist` is returned in the global variable `ibcnt`.

If none of the specified devices is requesting service, the error code ETAB is returned in `iberr`, and `ibcnt` contains the index of the `NOADDR` entry of the list.

If a device times out while responding to its serial poll, the error code EABO is returned in `iberr`, and the index of the timed-out device appears in `ibcnt`.

FindRQS (continued) FindRQS

Example:

Determine which one of the devices at addresses 8, 9, and 10 are requesting service.

BASICA

```
70  DIM addresslist%(4)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = 10
93  addresslist% (3) = NOADDR
100  CALL FindRQS (board%, addresslist% (0), result%)
```

Following this call, `result%` might contain the value `&H40` (the serial poll response), and `ibcnt` might contain the value 2, indicating that the device at `addresslist$ (2)` was the first device in the list found to be asserting SRQ.

QuickBASIC/BASIC

```
DIM addresslist%(4)
addresslist% (0) = 8
addresslist% (1) = 9
addresslist% (2) = 10
addresslist% (3) = NOADDR
CALL FindRQS (0, addresslist% (), result%)
```

C

```
unsigned short addresslist[3] = {8, 9, 10, NOADDR};
unsigned short result;
FindRQS (0, addresslist, &result);
```

GenerateREQF Generate REQF

Purpose: Cancel service request generated by GenerateREQT.

Format:

C

```
void GenerateREQF(short board, unsigned short addr);
```

`board` specifies a board number. `addr` specifies the 5-bit GPIB address of the simulated device that is no longer requesting service. The driver keeps track of the simulated devices currently requesting service so that it knows when to unassert the SRQ bus line.

Use this function when one of the simulated devices no longer needs service by the Controller-In-Charge. The `addr` parameter should be the address of the simulated device that no longer needs service. The driver unasserts the SRQ bus line, (if no other simulated devices are requesting service), and does not set the RSV bit when the simulated device is polled.

After a simulated device has been serial polled, the driver automatically performs a `GenerateREQF` on the polled device. Use this function if the simulated device has not yet been polled, but no longer requires servicing.

Example:

Refer to the function *GotoMultAddr* for a complete example of this function.

GenerateREQT Generate REQT

Purpose: Request service from the GPIB Controller-In-Charge.

Format:

C

```
void GenerateREQT(short board, unsigned short addr);
```

`board` specifies a board number. `addr` specifies the 5-bit GPIB address of the simulated device that is requesting service. The driver keeps track of the simulated devices currently requesting service, so that it knows when to unassert the SRQ bus line.

Use this function when one of the simulated devices needs service from the Controller-In-Charge. The `addr` parameter should be the address of the simulated device that needs service. The driver asserts the SRQ bus line. When the Controller notices that SRQ is asserted, it conducts a serial poll of the device. The `spollfunc` function then returns the appropriate serial poll response byte. The driver sets the RSV (Request Service) bit to 1 before sending the response byte to the Controller.

Example:

See the function *GotoMultAddr* for a complete example of this function.

GotoMultAddr GotoMultAddr

Purpose: Enable multiple primary or secondary address support.

Format:

C

```
void GotoMultAddr(short board,
                  unsigned short type,
                  unsigned short (_far _loadds
                      *addrfunc)(),
                  unsigned short (_far _loadds
                      *spollfunc)())
```

`board` specifies a board number. `type` must be either the constant `MultAddrPrimary` or `MultAddrSecondary` (defined in the language interface include file). Use this parameter to select one of two multiple address modes. A program simulates either multiple primary GPIB addresses or multiple secondary GPIB address of the same primary address. A program cannot simulate both.

`addrfunc` must be the address of the address selection function provided by the user. The driver calls this function whenever a primary or secondary (depending on `type`) GPIB address is on the bus. This function must determine if the given address is one of the simulated addresses and return TRUE (1) or FALSE (0) to the driver. See the discussion of the address selection function below for more information.

`spollfunc` must be the address of the serial poll response function provided by the user. The driver calls this function whenever one of the simulated devices is serial polled. This function should return the serial poll response byte for the given address. The driver then sends this response byte to the Controller. See the discussion of the serial poll response function below for more information.

Call this function once per interface board when you begin an application that simulates multiple GPIB addresses. This call places the driver in multiple primary or secondary address mode depending on the `type` field. To disable this mode, you must call the `ibonl` function with a value of 0 or 1. Your application *must* make a call to `ibonl` with a value of 0 before it terminates, because the driver maintains pointers to your application code,

GotoMultAddr (continued) GotoMultAddr

the `addrfunc` and `spollfunc` pointers. If your application exits without clearing these pointers, the driver might call those functions after they are removed from memory. This causes your computer to lock up.

Hardware interrupts on the interface board must be enabled for this function to work. If they are disabled, either through `IBCONF` or by an `ibconfig` function call, the call to this function returns the error `ECAP`. Refer to Chapter 3, *Understanding the NI-488.2 Software*, for a description of this error code.

After multiple address mode is enabled, the driver calls the `addrfunc` and `spollfunc` functions when information from the application is needed. Since these functions are called at interrupt time, you must take special care when writing them. Below is a list of rules that must be followed when writing functions that are called at interrupt time:

- Return from the interrupt call as soon as possible. Do not perform large calculations as this prevents the system from performing other interrupt activities, such as maintaining the system clock.
- Do not call any functions if you are not sure that they are re-entrant. This includes DOS and BIOS functions, standard C library function, Windows functions, and GPIB functions.
- Do not use a large amount of stack space. The stack provided to your function has about 512 bytes available for its use. You should also disable run-time stack overflow checking. The `/Gs` option of the Microsoft C compiler does this.
- If your application is written for Microsoft Windows 3, then you must ensure that your code and data segments are `FIXED` in memory. You can do this in the module definition file of your application.

The Address Selection Function:

The driver calls the address selection function whenever a primary or secondary GPIB address is present on the bus. The driver passes the GPIB address to the function. The function determines whether to accept

GotoMultAddr (continued) GotoMultAddr

the given address. If the function accepts the address, the interface board uses the given GPIB address. The application program can then read from or write to the bus as if it were the device at the given GPIB address.

Here is the function prototype of the address selection function:

```
unsigned short _far _loadds addrfunc(short board,
                                     unsigned short type,
                                     unsigned short addr)
```

The `_far` directive tells the compiler to generate a far return when this function exits. The `_loadds` directive tells the compiler to load the Data Segment register with the default data segment of the application. The function can now access the global variables of the application.

The driver passes the following parameters to the address selection function:

`board` is the index of the interface board on which the GPIB address is present. This parameter is only useful when your application uses multiple buses–that is, multiple interface boards.

`type` is the type of address that is present on the bus; either a talk address, a listen address, or a talk address while the board is in Serial Poll Mode State. These types are defined by the constants: `MultAddrListen`, `MultAddrTalk`, and `MultAddrSerialPoll` which are defined in the language interface include file.

`addr` is the 5-bit GPIB address that is currently present on the bus.

This function should return a non-zero value to accept the given address. It should return zero to reject the address.

Serial Poll Response Function:

The driver calls the serial poll response function when the interface board is serial polled by the GPIB Controller-In-Charge. The driver passes the GPIB address of the device being polled to this function. This function must return an 8-bit serial poll response byte which is sent to the Controller. If the simulated device is requesting service through the use of the

GotoMultAddr (continued) GotoMultAddr

`GenerateREQT` function at the time of the poll, the driver sets the Request Service (RSV) bit to 1 before sending the response byte to the Controller.

When a serial poll is conducted, the Controller sends out the talk address of the device it wants to poll. The address selection function is then called with `type` set to `MultAddrSerialPoll` just prior to calling the serial poll response function. Because the address selection function is called before this one, when the Controller tries to poll a device that is not simulated by the application program, the address selection function can return zero to reject the serial poll. This prevents the driver from calling the serial poll response function.

Here is the function prototype of the serial poll response function:

```
unsigned short  _far _loadds spollfunc(short board,
                                       unsigned short  addr)
```

The parameters passed to the function by the driver are:

`board` is the index of the interface board on which the GPIB address is present. This parameter is only useful when your application is using multiple buses–that is, multiple interface boards.

`addr` is the 5-bit GPIB address that is currently present on the bus.

This function should return the 8-bit serial poll response byte that is sent to the Controller.

Example:

C

The following example program simulates four GPIB devices. It simulates GPIB devices at primary addresses 1, 3, 24, and 30.

```
#include <stdio.h>
#include <string.h>
#include <process.h>
#include <bios.h>
#include "c:\at-gpib\c\decl.h"
```

GotoMultAddr (continued) GotoMultAddr

```
#define TRUE  1
#define FALSE 0

#define LAD  0x20  /* listen address mask */
#define TAD  0x40  /* talk address mask   */

#define BUFSIZE 512

/*
 *    Globals.
 */
short addressed = FALSE;
unsigned short address;
char buffer[ BUFSIZE + 2 ];

/*
 *  This function implements the "address
 *  selection" call-back function.  It it used
 *  to validate the addresses that are seen on
 *  the bus.  If GPIB addresses 1, 3, 24, or 30
 *  are seen on the bus, this function returns
 *  TRUE.  This means this application should
 *  simulate those devices.
 */
unsigned short _far _loadds addrfunc(short board,
                              unsigned short type,
                              unsigned short addr)
{
   if ((addr == 1) || (addr == 3) ||
       (addr == 24) || (addr == 30)) {
      /*
       * If the program is about to be serial polled then
       * accept the address, so that the "spollfunc"
       * is called to return the serial poll response
       * byte.
       */
      if (type == MultAddrSerialPoll) {
         return (TRUE);
      }
      /*
       * If this is a listen address, then set the
       * global "addressed" to TRUE and store the
       * listen address in the global "address".
       */
```

GotoMultAddr (continued) GotoMultAddr

```
      else if (type == MultAddrListen) {
         addressed = TRUE;
         address = (LAD | addr);
         return (TRUE);
      }
      /*
       * If this is a talk address, then set the
       * global "addressed" to TRUE and store the
       * talk address in the global "address".
       */
      else if (type == MultAddrTalk) {
         addressed = TRUE;
         address = (TAD | addr);
         return (TRUE);
      }
   }

   /* Return FALSE since you do not claim this
    * address.
    */
   return (FALSE);

}  /* end of addrfunc */

/*
 * This function implements the "Serial Poll Response"
 * call-back function. It always returns the GPIB
 * address of the simulated device as the serial poll
 * response byte.
 */
unsigned short _far _loadds spollfunc(short board,
                                unsigned short addr)
{
   return (addr);
}

/*
 * This program is an example of how to simulate
 * multiple GPIB addresses. The program waits in a
 * loop until one of its simulated addresses is
 * present on the bus.  It then reads data or writes
 * data for the simulated device.  If you press any
 * key, the program terminates.
 */
short _cdecl main ( void )
```

GotoMultAddr (continued) GotoMultAddr

```
{
   short testing;
   short SimulatedAddress;

   addressed = FALSE;
   testing = TRUE;

   /*
    * Enable multiple primary GPIB addresses for
    * interface board #0. Pass the address of the
    * "address selection" function (addrfunc) and the
    * "serial poll response" function (spollfunc).
    */
   GotoMultAddr(0, MultAddrPrimary, addrfunc,
                spollfunc);
   if (ibsta & ERR) {
      printf("Error calling GotoMultAddr.\n");
      ibonl(0, 0);
      exit(1);
   }

   /*
    * This is the main loop. Stay here until any
    * key is pressed on the keyboard.
    */
   while (testing) {
      printf("\nWaiting to be addressed....\n");
      /*
       * Check for any key to be pressed.
       */
      while (addressed == FALSE) {
         if (_bios_keybrd(_KEYBRD_READY)) {
            testing = FALSE;
            break;
         }
      }

      addressed = FALSE;
      SimulatedAddress = address;

      /*
       * As long as you did not press a key to exit,
       * then the program must be addressed to talk or
       * listen.
       */
```

GotoMultAddr (continued) GotoMultAddr

```
if (testing == TRUE) {
   /*
    * If the address is a listen address, then
    * read in data byte for the simulated
    * device. After reading in the bytes, call
    * GenerateREQT to request service for the
    * simulated device.
    */
   if ((SimulatedAddress & (LAD | TAD)) == LAD) {
      printf("Address %d is listening.\n",
             (SimulatedAddress & ~LAD));

      /*
       * Read a buffer for the given device.
       */
      RcvRespMsg(0, buffer,
            (unsigned long)BUFSIZE, STOPend);
      if (ibsta & ERR) {
         printf("Error from RcvRespMsg.\n");
         ibonl(0, 0);
         exit(1);
      }

      /*
       * Put a NUL byte at the end of the buffer
       * and call printf to output the buffer
       * to the screen.
       */
      buffer[ibcntl] = '\0';
      printf("Received ‘%s’ for PAD %d\n",
              buffer,
              (SimulatedAddress & ~LAD));

      /*
       * Now assert SRQ to request service for
       * the simulated device.
       */
      GenerateREQT(0,(SimulatedAddress & ~LAD));
   }
   /*
    * If the address in a talk address, then
    * output a buffer containing the GPIB
    * address of the simulated device.  Then
```

GotoMultAddr (continued) GotoMultAddr

```
         * call GenerateREQF for the simulated
         * device.  It no longer requires service
         * from the Controller.
         */
        else if ((SimulatedAddress & (LAD | TAD))
                       == TAD) {
           printf("Address %d talking.\n",
                   (SimulatedAddress & ~TAD));

           sprintf(buffer,
              "Data from GPIB address %d.",
              (SimulatedAddress & ~TAD));

           SendDataBytes(0, buffer,
              (unsigned long)strlen(buffer), DABend);
           if (ibsta & ERR) {
              printf("Error from SendDataBytes.\n");
              ibonl(0, 0);
              exit(1);
           }

           GenerateREQF(0,(SimulatedAddress & ~TAD));
        }
        else {
           printf("NOT talk or listen addressed.\n");
           ibonl(0, 0);
           exit(1);
        }
     }
   }

   /*
    * Be certain that you call ibonl with a value
    * of 0 before exiting the program.
    */
   ibonl(0, 0);

   return 0;

}  /* end of main */
```

PassControl PassControl

Purpose: Pass control to another device with Controller capability.

Format:

BASICA/QuickBASIC/BASIC

```
CALL PassControl (board%, address%)
```

C

void `PassControl` (short `board,` unsigned short `address)`

`board` specifies a board number. The GPIB Device Take Control message is sent to the device at the given address. The parameter `address` contains in its low byte the primary GPIB address of the device to be passed control. The high byte should be 0 if the device has no secondary address. Otherwise, it should contain the desired secondary address.

Example:

Pass control to a Controller connected to board `0` whose primary GPIB address is 9.

BASICA/QuickBASIC/BASIC

```
80  board% = 0
90  address% = 9
100  CALL PassControl (board%, address%)
```

C

```
PassControl (0, 9);
```

PPoll PPoll

Purpose: Perform a parallel poll.

Format:

BASICA/QuickBASIC/BASIC

```
CALL PPoll (board%, result%)
```

C

void `PPoll` (short `board`, unsigned short `result`)

`board` specifies a board number. A parallel poll is conducted, and the eight-bit result is stored into `result`. Only the lower eight bits of `result` are affected. In BASICA, QuickBASIC, and BASIC, the upper byte contains the same value it did before the call was made.

Each bit of the poll result returns one bit of status information from each device that has been configured for parallel polls. The state of each bit (0 or 1), and the interpretation of these states are based on the latest parallel poll configuration sent to the devices and the individual status of the devices.

Example:

Perform a parallel poll on board `0`.

BASICA/QuickBASIC/BASIC

```
80  board% = 0
100  CALL PPoll (board%, result%)
```

C

```
unsigned short result;
PPoll (0, &result);
```

PPollConfig PPollConfig

Purpose: Configure a device for parallel polls.

Format:

BASICA/QuickBASIC/BASIC

```
CALL PPollConfig (board%, address%, dataline%, sense%)
```

C

```
void PPollConfig (short board, unsigned short address,
                  unsigned short dataline, unsigned short sense)
```

`board` specifies a board number. The GPIB device at `address` is configured for parallel polls according to the `dataline` and `sense` parameters. `dataline` is the data line (1-8) on which the device is to respond, and `sense` indicates the condition under which the data line is to be asserted or unasserted. The device is expected to compare this sense value (0 or 1) to its individual status bit, and respond accordingly.

Devices have the option of configuring themselves for parallel polls, in which case they are to ignore attempts by the Controller to configure them. You should determine whether the device is locally or remotely configurable before using `PPollConfig` or `PPollUnconfig`.

Example:

Configure a device connected to board 0 at address 8 so that it responds to parallel polls on data line 5 with sense 0 (assert the line if the individual status is 0, unassert the line if the individual status is 1).

BASICA/QuickBASIC/BASIC

```
70  address% = 8
80  board% = 0
90  dataline% = 5
91  sense% = 0
100  CALL PPollConfig (board%, address%, dataline%, sense%)
```

C

```
PPollConfig (0, 8, 5, 0);
```

PPollUnconfig PPollUnconfig

Purpose: Unconfigure devices for parallel polls.

Format:

BASICA

```
CALL PPollUnconfig (board%, addresslist% (0))
```

QuickBASIC/BASIC

```
CALL PPollUnconfig (board%, addresslist% ())
```

C

```
void PPollUnconfig (short board, unsigned short addresslist [])
```

`board` specifies a board number. The GPIB devices whose addresses are contained in the address array are unconfigured for parallel polls; that is, they no longer participate in polls. The parameter `addresslist` is an array of address integers of any size, terminated by the value `NOADDR`.

If the array contains only the value `NOADDR`, or is `NULL` in C, the GPIB Parallel Poll Unconfigure (PPU) message is sent, unconfiguring all devices.

Example:

Unconfigure two devices connected to board 0 whose GPIB addresses are 8 and 9.

BASICA

```
70  DIM addresslist%(3)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = NOADDR
100  CALL PPollUnconfig (board%, addresslist% (0))
```

PPollUnconfig (continued) PPollUnconfig

QuickBASIC/BASIC

```
DIM addresslist%(3)
addresslist% (0) = 8
addresslist% (1) = 9
addresslist% (2) = NOADDR
CALL PPollUnconfig (0, addresslist% ())
```

C

```
unsigned short addresslist[3] = {8, 9, NOADDR};
PPollUnconfig (0, addresslist);
```

RcvRespMsg RcvRespMsg

Purpose: Read data bytes from already addressed device.

Format:

BASICA/QuickBASIC/BASIC

```
CALL RcvRespMsg (board%, data$, termination%)
```

C

```
void RcvRespMsg (short board, unsigned char data [],
                 unsigned long count, unsigned short termination)
```

`board` specifies a board number. Up to `count` data bytes are read from the GPIB and placed into the pre-allocated string `data`. In BASICA, QuickBASIC, and BASIC, the amount of data is inferred from the length of the string, which must be pre-allocated to a suitable length. The `count` argument in C is of type `long`; however, `integer` values and variables may also be passed. `termination` is a flag used to describe the method of signaling the end of the data. If it is a value between 0 and hex 00FF, the ASCII character with the corresponding hex value is considered the termination character, and the read is stopped when the character is detected. If `termination` is the constant STOPend (defined in the header file `DECL.BAS`, `QBDECL.BAS`, `MBDECL.BAS`, or `DECL.H`), then the read is stopped when EOI is detected.

`RcvRespMsg` assumes that the GPIB Talker and Listeners have already been addressed by a prior call to routines such as `ReceiveSetup`, `Receive`, or `SendCmds`. Thus, it is used specifically to skip the addressing step of GPIB management. The `Receive` routine is normally used to accomplish the entire sequence of addressing followed by the reception of data bytes.

RcvRespMsg (continued) RcvRespMsg

Example:

Receive 100 bytes from an already addressed Talker. The transmission should be terminated when a linefeed character is detected.

BASICA

```
80  board% = 0
90  data$ = space$(100)
91  termination% = &H0A
100  CALL RcvRespMsg (board%, data$, termination%)
```

QuickBASIC/BASIC

```
DIM data AS STRING*100
CALL RcvRespMsg (0, data$, &H0A)
```

C

```
unsigned char data[100];
RcvRespMsg (0, data, 100, '\n');
```

ReadStatusByte ReadStatusByte

Purpose: Serial poll a single device to get its status byte.

Format:

BASICA/QuickBASIC/BASIC

```
CALL ReadStatusByte (board%, address%, result%)
```

C

```
void ReadStatusByte (short board, unsigned short address,
                     unsigned short *result)
```

`board` specifies a board number. The indicated device is serial polled, and its status byte is placed into the variable `result`, with the status byte zero-extended into the upper byte.

Example:

Serial poll the device at address 8 and return its status byte.

BASICA/QuickBASIC/BASIC

```
80  board% = 0
90  address% = 8
100  CALL ReadStatusByte (board%, address%, result%)
```

C

```
unsigned short result;
ReadStatusByte (0, 8, &result);
```

Receive Receive

Purpose: Read data bytes from a GPIB device.

Format:

BASICA/QuickBASIC/BASIC

```
CALL Receive (board%, address%, data$, termination%)
```

C

```
void Receive (short board, unsigned short address, unsigned char data [],
              unsigned long count, unsigned short termination)
```

`board` specifies a board number. The indicated GPIB device is addressed, and up to `count` data bytes are read from that device and placed into the pre-allocated string `data.` The `count` value is an `integer` in BASICA, but is of type long in QuickBASIC, BASIC, and C. Even though it is a long value in these languages, however, integer values and variables may also be passed. `termination` is a value used to describe the method of signaling the end of the data. If it is a value between 0 and hex 00FF, the ASCII character with the corresponding hex value is considered the termination character, and the read is stopped when the character is detected. If `termination` is the constant STOPend (defined in the header file `DECL.BAS`, `QBDECL.BAS`, `MBDECL.BAS`, or `DECL.H`), the read is stopped when END is detected.

Example:

Receive 100 bytes from the device at address 8. The transmission should be terminated when END is detected.

BASICA/QuickBASIC/BASIC

```
80  board% = 0
81  address% = 8
90  data$ = space$(100)
91  termination% = STOPend
100  CALL Receive (board%, address%, data$, termination%)
```

Receive (continued) Receive

C

```
unsigned char data[100];
Receive (0, 8, data, 100, STOPend);
```

ReceiveSetup ReceiveSetup

Purpose: Prepare a particular device to send data bytes and prepare the GPIB interface board to read them.

Format:

BASICA/QuickBASIC/BASIC

```
CALL ReceiveSetup (board%, address%)
```

C

void `ReceiveSetup` (short `board`, unsigned short `address`)

`board` specifies a board number. The indicated GPIB device is addressed as a Talker, and the indicated board is addressed as a Listener. Following this routine, it is common to call a routine such as `RcvRespMsg` to actually transfer the data from the Talker.

This routine is useful to initially address devices in preparation for receiving data, followed by multiple calls of `RcvRespMsg` to receive multiple blocks of data, thus eliminating the need to readdress the devices between blocks. Alternatively, the `Receive` routine could be used to send the first data block, followed by `RcvRespMsg` for all the subsequent blocks.

ReceiveSetup (continued) ReceiveSetup

Example:

Prepare a GPIB device at address 8 to send data bytes to board `0`. Then, receive messages of up to 100 bytes from the device, and store it in a string. The message is to be terminated with END.

BASICA

```
80  board% = 0
90  address = 8
94  messages$ = space$(100)
100  CALL ReceiveSetup (board%, address%)
130  termination% = STOPEND
140  CALL RcvRespMsg (board%, messages$, termination%)
```

QuickBASIC/BASIC

```
DIM message AS STRING*100
CALL ReceiveSetup (0, 80)
CALL RcvRespMsg (0, message$, STOPEND)
```

C

```
unsigned char message[100];
ReceiveSetup (0, 8)
RcvRespMsg (0, message, 100, STOPEND);
```

ResetSys ResetSys

Purpose: Initialize a GPIB system on three levels.

Format:

BASICA

```
CALL ResetSys (board%, addresslist%(0))
```

QuickBASIC/BASIC

```
CALL ResetSys (board%, addresslist%())
```

C

```
void ResetSys (short board, unsigned short adddresslist [])
```

`board` specifies a board number. The GPIB system is initialized on the following three levels:

• Bus initialization:	Remote Enable (REN) is asserted, followed by Interface Clear (IFC), causing all devices to become unaddressed and the GPIB interface board (the System Controller) to become the Controller-in-Charge.
• Message exchange initialization:	The Device Clear (DCL) message is sent to all connected devices. This ensures that all 488.2 compatible devices can receive the Reset (RST) message that follows.
• Device initialization:	`*RST` message is sent to all devices whose addresses are contained in the `addresslist` argument. This causes device-specific functions within each device to be initialized.

ResetSys (continued) ResetSys

Example:

Completely reset a GPIB system containing devices at addresses 8, 9, and 10.

BASICA

```
70  DIM addresslist%(4)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = 10
93  addresslist% (3) = NOADDR
100  CALL ResetSys (board%, addresslist%(0))
```

QuickBASIC/BASIC

```
DIM addresslist%(4)
addresslist% (0) = 8
addresslist% (1) = 9
addresslist% (2) = 10
addresslist% (3) = NOADDR
CALL ResetSys (0, addresslist%())
```

C

```
unsigned short addresslist[4] = {8, 9, 10,
NOADDR};
ResetSys (0, addresslist);
```

Send

Purpose: Send data bytes to a single GPIB device.

Format:

BASICA/QuickBASIC/BASIC

```
CALL Send (board%, address%, data$, eotmode%)
```

C

```
void Send  (short board,  unsigned short address,
            unsigned char data [], unsigned long count,
            unsigned short eotmode)
```

`board` specifies a board number. The indicated GPIB device is addressed as a Listener, the indicated board is addressed as a Talker, and `count` data bytes contained in `data` are sent. The `count` value is an integer in BASICA, but is of type long in QuickBASIC, BASIC, and C. Even though it is a long value in these languages, however, integer values and variables may also be passed. `eotmode` is a flag used to describe the method of signaling the end of the data to the Listener. It should be set to one of the following constants:

• `NLend`	Send NL (linefeed) with EOI after the data bytes.
• `DABend`	Send EOI with the last data byte in the string.
• `NULLend`	Do nothing to mark the end of the transfer.

These constants are defined in the header files `DECL.BAS`, `QBDECL.BAS`, `MBDECL.BAS`, and `DECL.H`.

Send (continued) Send

Example:

Send an identification query to the GPIB device at address 8. Terminate the transmission using a linefeed character with END.

BASICA

```
80  board% = 0
81  address% = 8
90  data$ = "*IDN?"
91  eotmode% = NLend
100  CALL Send (board%, address%, data$, eotmode%)
```

QuickBASIC/BASIC

```
CALL Send (0, 8, "*IDN?", NLend)
```

C

```
Send (0, 8, "*IDN?", 5, NLend);
```

SendCmds SendCmds

Purpose: Send GPIB command bytes.

Format:

BASICA/QuickBASIC/BASIC

```
CALL SendCmds (board%, commands$)
```

C

```
void SendCmds (short board, unsigned char commands [],
               unsigned long count)
```

`board` specifies a board number. `commands` contains command bytes to be sent onto the GPIB. The number of bytes to be sent from the string is indicated by the argument `count`. The `count` value is an integer in BASICA, but is of type long in QuickBASIC, BASIC, and C. Even though it is a long value in these languages, however, integer values and variables may also be passed.

`SendCmds` is not normally required for GPIB operation. It is to be used when specialized command sequences, which are not provided for in other routines, must be sent onto the GPIB.

Example:

Controller, at address 0, simultaneously triggers GPIB devices at addresses 8 and 9, and immediately places them into local mode.

BASICA

```
80  board% = 0
90  commands$ = chr$(&3F)+chr$(&40)+chr$(&H28)+chr$(&H29)
+chr$(&H04)+chr$(&H01)
100  CALL SendCmds (board%, commands$)
```

QuickBASIC/BASIC

```
CALL SendCmds (0, chr$(&3F)+chr$(&40)+chr$(&H28)+chr$(&H29)
+chr$(&H04)+chr$(&H01))
```

SendCmds (continued) SendCmds

C

```
SendCmds (0, "\x3F\x40\x28\x29\x04\x01", 6);
```

SendDataBytes SendDataBytes

Purpose: Send data bytes to already addressed devices.

Format:

BASICA/QuickBASIC/BASIC

```
CALL SendDataBytes (board%, data$, eotmode%)
```

C

```
void SendDataBytes (short board, unsigned char data [],
                    unsigned long count, unsigned short eotmode)
```

`board` specifies a board number. `data` contains data bytes to be sent on to the GPIB. The number of bytes to be sent from the string is indicated by the argument `count`. The `count` value is an integer in BASICA, but is of type long in QuickBASIC, BASIC, and C. Even though it is a long value in these languages, however, integer values and variables may also be passed. `eotmode` is a flag used to describe the method of signaling the end of the data to the Listeners. It should be set to one of the following constants:

• `NLend`	Send NL (linefeed) with EOI after the data bytes.
• `DABend`	Send EOI with the last data byte in the string.
• `NULLend`	Do nothing to mark the end of the transfer.

These constants are defined in the header files `DECL.BAS`, `QBDECL.BAS`, `MBDECL.BAS`, and `DECL.H.`

`SendDataBytes` assumes that all GPIB Listeners have already been addressed by a prior call to functions such as `SendSetup`, `Send`, or `SendCmds`. Thus, it is used specifically to skip the addressing step of GPIB management. The `Send` routine is normally used to accomplish the entire sequence of addressing followed by the transmission of data bytes.

SendDataBytes (continued) SendDataBytes

Example:

Send an identification query to all addressed Listeners. The transmission should be terminated with a linefeed character with END.

BASICA

```
80  board% = 0
90  data$ = “*IDN?”
91  eotmode% = NLend
100  CALL SendDataBytes (board%, data$, eotmode%)
```

QuickBASIC/BASIC

```
CALL SendDataBytes (0, “*IDN?”, NLend)
```

C

```
SendDataBytes (0, “*IDN?”, 5, NLend);
```

SendIFC SendIFC

Purpose: Clear the GPIB interface functions with IFC.

Format:

BASICA/QuickBASIC/BASIC

```
CALL SendIFC (board%)
```

C

void `SendIFC` (short `board`)

`board` specifies a board number. The GPIB Device IFC message is issued, resulting in the interface functions of all connected devices returning to their cleared states.

This function is used as part of GPIB initialization. It forces the GPIB interface board to be Controller of the GPIB, and ensures that the connected devices are all unaddressed and that the interface functions of the devices are in their idle states.

Example:

Clear the interface functions of the devices connected to board `0`.

BASICA

```
80  board% = 0
100  CALL SendIFC (board%)
```

QuickBASIC/BASIC

```
CALL SendIFC (0)
```

C

```
SendIFC (0);
```

SendList SendList

Purpose: Send data bytes to multiple GPIB devices.

Format:

BASICA

```
CALL SendList (board%, addresslist%(0), data$, eotmode%)
```

QuickBASIC/BASIC

```
CALL SendList (board%, addresslist%(), data$, eotmode%)
```

C

```
void SendList (short board, unsigned short addresslist [],
               unsigned char data [],  unsigned long count,
               unsigned short eotmode)
```

`board` specifies a board number. `addresslist` contains a list of primary GPIB addresses, terminated by the value `NOADDR`. The GPIB devices whose addresses are contained in the address array are addressed as Listeners, the indicated board is addressed as a Talker, and `count` data bytes contained in `data` are sent. The `count` value is an integer in BASICA, but is of type long in QuickBASIC, BASIC, and C. Even though it is a long value in these languages, however, integer values and variables may also be passed. `eotmode` is a flag used to describe the method of signaling the end of the data to the Listener. It should be set to one of the following constants:

• `NLend`	Send NL (linefeed) with EOI after the data bytes.
• `DABend`	Send EOI with the last data byte in the string.
• `NULLend`	Do nothing to mark the end of the transfer.

These constants are defined in the header files `DECL.BAS`, `QBDECL.BAS`, `MBDECL.BAS`, and `DECL.H.`

This routine is similar to `Send`, except that multiple Listeners are able to receive the data with only one transmission.

SendList (continued) SendList

Example:

Send an identification query to the GPIB devices at address 8 and 9. The transmission should be terminated using a linefeed character with EOI.

BASICA

```
70  DIM addresslist%(3)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = NOADDR
93  data$ = "*IDN?"
94  eotmode% = NLend
100  CALL SendList (board%, addresslist%(0), data$, eotmode%)
```

QuickBASIC/BASIC

```
DIM addresslist%(3)
addresslist% (0) = 8
addresslist% (1) = 9
addresslist% (2) = NOADDR
CALL SendList (0, addresslist%(), "*IDN?", NLend)
```

C

```
unsigned addresslist[3] = {8, 9, NOADDR};
SendList (0, addresslist, "*IDN?", 5, NLend);
```

SendLLO SendLLO

Purpose: Send the Local Lockout message to all devices.

Format:

BASICA/QuickBASIC/BASIC

```
CALL SendLLO (board%)
```

C

```
void SendLLO (short board)
```

`board` specifies a board number. The GPIB Local Lockout message is sent to all devices, so that the devices cannot independently choose the local or remote states. While Local Lockout is in effect, only the Controller can alter the local or remote state of the devices by sending appropriate GPIB messages.

`SendLLO` is reserved for use in unusual local/remote situations, particularly those in which all devices are to be locked into local programming state. In the typical case of placing devices in Remote Mode With Lockout state, the `SetRWLS` routine should be used.

Example:

Send the Local Lockout message to all devices connected to board 0.

BASICA

```
80  board% = 0
100  CALL SendLLO (board%)
```

QuickBASIC/BASIC

```
CALL SendLLO (0)
```

C

```
SendLLO (0);
```

SendSetup SendSetup

Purpose: Prepare particular devices to receive data bytes.

Format:

BASICA

```
CALL SendSetup (board%, addresslist% (0))
```

QuickBASIC/BASIC

```
CALL SendSetup (board%, addresslist% ())
```

C

```
void SendSetup (short board, unsigned short addresslist [])
```

`board` specifies a board number. The GPIB devices whose addresses are contained in the `addresslist` array are addressed as Listeners, and the indicated board is addressed as a Talker. Following this call, it is common to call a routine such as `SendDataBytes` to actually transfer the data to the Listeners. The parameter `addresslist` is an array for any size of address integers, terminated by the value `NOADDR`.

This command would be useful to initially address devices in preparation for sending data, followed by multiple calls of `SendDataBytes` to send multiple blocks of data, thus eliminating the need to readdress the devices between blocks. Alternatively, the `Send` routine could be used to send the first data block, followed by `SendDataBytes` for all the subsequent blocks.

SendSetup (continued) SendSetup

Example:

Prepare GPIB devices at addresses 8 and 9 to receive data bytes. Then, send both devices the five messages stored in a string array. EOI is to be sent along with the last byte of the last message.

BASICA

```
70  DIM addresslist%(3)
75  DIM messages$(5)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = NOADDR
94  messages$(0) = "Message 0"
95  messages$(1) = "Message 1"
96  messages$(2) = "Message 2"
97  messages$(3) = "Message 3"
98  messages$(4) = "Message 4"
100  CALL SendSetup (board%, addresslist% (0))
110  for i% = 0 to 3
120    CALL SendDataBytes (board%, messages$(i%),NULLEND)
130  NEXT i%
140  CALL SendDataBytes (board%, messages$(4), NLend)
```

QuickBASIC/BASIC

```
DIM addresslist%(3)
DIM messages$(5)
addresslist% (0) = 8
addresslist% (1) = 9
addresslist% (2) = NOADDR
messages$(0) = "Message 0"
messages$(1) = "Message 1"
messages$(2) = "Message 2"
messages$(3) = "Message 3"
messages$(4) = "Message 4"
CALL SendSetup (0, addresslist% ())
FOR i% = 0 to 3
  CALL SendDataBytes (board%, messages$(i%), NULLEND)
NEXT i%
CALL SendDataBytes (board%, messages$(4), NLend)
```

SendSetup (continued) SendSetup

C

```
int i;
unsigned short addresslist[3] = {8, 9, NOADDR};
unsigned char *messages[5] = {
     "Message 0",
     "Message 1",
     "Message 2",
     "Message 3",
     "Message 4"  };
SendSetup (0, addresslist)
for (i = 0; i < 4; i++)
     SendDataBytes (0, messages[i],
                     strlen (messages[i]),NULLend);
SendDataBytes (0, messages[4], strlen (messages[4])NLend);
```

SetRWLS SetRWLS

Purpose: Place particular devices in the Remote With Lockout State.

Format:

BASICA

```
CALL SetRWLS (board%, addresslist% (0))
```

QuickBASIC/BASIC

```
CALL SetRWLS (board%, addresslist% ())
```

C

void `SetRWLS` (short `board`, unsigned short `addresslist` [])

`board` specifies a board number. The GPIB devices whose addresses are contained in the `addresslist` array are placed in remote mode by asserting Remote Enable (REN) and addressing the devices as Listeners. In addition, all devices are placed in Lockout State, which prevents them from independently returning to local programming mode without passing through the Controller. The parameter `addresslist` is an array of any size of address integers, terminated by the value `NOADDR`.

Example:

Place the devices at GPIB addresses 8 and 9 in Remote With Lockout State.

BASICA

```
70  DIM addresslist%(3)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = NOADDR
100  CALL SetRWLS (board%, addresslist% (0))
```

SetRWLS (continued) SetRWLS

QuickBASIC/BASIC

```
DIM addresslist%(3)
addresslist% (0) = 8
addresslist% (1) = 9
addresslist% (2) = NOADDR
CALL SetRWLS (0, addresslist% ())
```

C

```
unsigned short addresslist[3] = {8, 9, NOADDR};
SetRWLS (0, addresslist);
```

TestSRQ TestSRQ

Purpose: Determine the current state of the SRQ line.

Format:

BASICA/QuickBASIC/BASIC

```
CALL TestSRQ (board%, result%)
```

C

```
void TestSRQ (short board, short *result)
```

`board` specifies a board number. This call places the value 1 in the variable `result` if the GPIB SRQ line is asserted. Otherwise, it places the value of 0 into `result`.

This routine is similar in format to the `WaitSRQ` routine, except that `WaitSRQ` suspends itself waiting for an occurrence of SRQ, whereas `TestSRQ` returns immediately with the current SRQ state.

Example:

Determine the current state of SRQ.

BASICA

```
80  board% = 0
100  CALL TestSRQ (board%, result%)
105  IF result% = 1 then
110      'SRQ is asserted
111  ELSE
112      'No SRQ at this time
113  END IF
```

TestSRQ **(continued)** **TestSRQ**

QuickBASIC/BASIC

```
CALL TestSRQ (0, result%)
IF result% = 1 then
    ' SRQ is asserted
ELSE
    ' No SRQ at this time
END IF
```

C

```
short result;
TestSRQ (0, &result);
if (result == 1)
  {  /* SRQ is asserted */  }
else
  {  /* No SRQ at this time */  }
```

TestSys TestSys

Purpose: Cause devices to conduct self-tests.

Format:

BASICA

```
CALL TestSys (board%, addresslist% (0), resultlist% (0))
```

QuickBASIC/BASIC

```
CALL TestSys (board%, addresslist% (), resultlist% ())
```

C

```
void TestSys  (short board, unsigned short addresslist [],
                unsigned short resultlist [])
```

`board` specifies a board number. The GPIB devices whose addresses are contained in the address array are simultaneously sent a message that instructs them to conduct their self-test procedures. Each device returns an integer code signifying the results of its tests, and these codes are placed into the corresponding elements of the `resultlist` array. The IEEE-488.2 standard specifies that a result code of 0 indicates that the device passed its tests, and any other value indicates that the tests resulted in an error. The variable `ibcnt` contains the number of devices that failed their tests. The parameter `addresslist` is an array of address integers of any size, terminated by the value `NOADDR`.

TestSys (continued) TestSys

Example:

Instruct two devices connected to board `0` whose GPIB addresses are 8 and 9 to perform their self-tests.

BASICA/QuickBASIC/BASIC

```
70  DIM addresslist%(3)
71  DIM resultlist%(2)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = NOADDR
100  CALL TestSys (board%, addresslist% (0),
resultlist%(0))
101  ' If any of the results are non-zero, the
102  ' corresponding device has failed the test.
```

C

```
unsigned short addresslist[3] = {8, 9, NOADDR};
unsigned short resultlist[2];
TestSys (0, addresslist, resultlist);
```

Trigger Trigger

Purpose: Trigger a single device.

Format:

BASICA/QuickBASIC/BASIC

```
CALL Trigger (board%, address%)
```

C

```
void Trigger (short board, unsigned short address)
```

`board` specifies a board number. The GPIB Group Execute Trigger message is sent to the device at the given address. The parameter `address` contains in its low byte the primary GPIB address of the device to be cleared. The high byte should be 0 if the device has no secondary address. Otherwise, it should contain the desired secondary address. If the address is `NOADDR`, the Group Execute Trigger message is sent with no addressing, thereby triggering all previously addressed Listeners.

The `Trigger` routine is used to trigger exactly one GPIB device. To send a single message that triggers several particular GPIB devices, use the `TriggerList` function.

Example:

Trigger a digital voltmeter connected to board `0` whose primary GPIB address is 9 and whose secondary GPIB address is 97.

BASICA

```
80  board% = 0
90  address% = 9 + 256*97
100  CALL Trigger (board%, address%)
```

Trigger (continued) Trigger

QuickBASIC/BASIC

```
CALL Trigger (0, 9 + 256*97)
```

C

```
Trigger (0, MakeAddr (9, 97));

/* In C, a macro has been defined in the header
 * file DECL.H, MakeAddr(p,s), which can be
 * used to pack the primary and secondary
 * addresses into the correct form.
 */
```

TriggerList TriggerList

Purpose: Trigger multiple devices.

Format:

BASICA

```
CALL TriggerList (board%, addresslist% (0))
```

QuickBASIC/BASIC

```
CALL TriggerList (board%, addresslist% ())
```

C

```
void TriggerList (short board, unsigned short addresslist [])
```

`board` specifies a board number. The GPIB devices whose addresses are contained in the address array are triggered simultaneously. The parameter `addresslist` is an array of address integers of any size, terminated by the value `NOADDR`. If the array contains only the value `NOADDR`, or if in C, a `NULL` value is passed, the Group Execute Trigger message is sent without addressing, thereby triggering all previously addressed Listeners.

Although the `TriggerList` routine is general enough to trigger any number of GPIB devices, the `Trigger` function should be used in the common case of triggering exactly one GPIB device.

TriggerList (continued) TriggerList

Example:

Trigger simultaneously two devices connected to board `0` whose GPIB addresses are 8 and 9.

BASICA

```
70  DIM addresslist%(3)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = NOADDR
100  CALL TriggerList (board%, addresslist% (0))
```

QuickBASIC/BASIC

```
DIM addresslist%(3)
addresslist% (0) = 8
addresslist% (1) = 9
addresslist% (2) = NOADDR
CALL TriggerList (0, addresslist% ())
```

C

```
unsigned short addresslist[3] = {8, 9, NOADDR};
TriggerList (0, addresslist);
```

WaitSRQ WaitSRQ

Purpose: Wait until a device asserts Service Request.

Format:

BASICA/QuickBASIC/BASIC

```
CALL WaitSRQ (board%, result%)
```

C

```
void WaitSRQ (short board, short *result)
```

`board` specifies a board number. This routine is used to suspend execution of the program until a GPIB device connected to the indicated board asserts the Service Request (SRQ) line. If the SRQ occurs within the timeout period, the variable `result` is set to the value 1. If no SRQ is detected before the timeout period expires, `result` is set to 0.

Notice that this call is similar in format to the `TestSRQ` routine, except that `TestSRQ` returns immediately with SRQ status, whereas `WaitSRQ` suspends the program for, at most, the duration of the timeout period waiting for an SRQ to occur.

Example:

Wait for a GPIB device to request service, and then determine which of three devices at addresses 8, 9, and 10 requested the service.

BASICA

```
70  DIM addresslist%(4), resultlist%(3)
80  board% = 0
90  addresslist% (0) = 8
91  addresslist% (1) = 9
92  addresslist% (2) = 10
93  addresslist% (3) = NOADDR
100  CALL WaitSRQ (board%, result%)
105  IF result% = 1 THEN
110      CALL AllSpoll (board%, addresslist% (0),
resultlist%(0))
120  END IF
130  ' resultlist%() now contains the serial
131  ' poll responses for the three devices.
```

WaitSRQ (continued) WaitSRQ

QuickBASIC/BASIC

```
DIM addresslist%(4), resultlist%(3)
addresslist% (0) = 8
addresslist% (1) = 9
addresslist% (2) = 10
addresslist% (3) = NOADDR
CALL WaitSRQ (0, result%)
IF result% = 1 then
    CALL AllSpoll (0, addresslist% (), resultlist%())
END IF
```

C

```
unsigned short addresslist[4] = {8, 9, 10, NOADDR};
unsigned short resultlist[3];
short result;
WaitSRQ (0, &result);
if (result == 1)
    AllSpoll (0, addresslist, resultlist);
```

BASICA/QuickBASIC/BASIC/C GPIB Programming Examples

You can take full advantage of the ANSI/IEEE 488.2-1987 standard by using the NI-488.2 routines. These routines are completely compatible with the Controller commands and protocols defined in IEEE-488.2.

The NI-488.2 routines are easy to learn and use. Only a few routines are needed for most application programs.

These examples illustrate the programming steps that could be used to program a representative IEEE-488.2 instrument from your personal computer using the NI-488.2 routines. The applications are written in BASICA, QuickBASIC, BASIC, and C. The target instrument is a digital voltmeter (DVM). This instrument is otherwise unspecified (that is, it is not a DVM manufactured by any particular manufacturer). The purpose here is to explain how to use the driver to execute NI-488.2 programming and control sequences and not how to determine those sequences.

Note: For a more detailed description of each step, refer to *Writing an Advanced Program Using NI-488.2 Routines*, in the Getting Started manual that you received with your interface board.

1. Load in the definitions of the NI-488.2 routines from a file that is on your distribution diskette.

2. Initialize the IEEE-488 bus and the interface board Controller circuitry so that the IEEE-488 interface for each device is quiescent, and so that the interface board is Controller-In-Charge and is in the Active Controller State (CACS).

3. Find all of the Listeners:

   a. Find all of the instruments attached to the IEEE-488 bus.

   b. Create an array that contains all of the IEEE-488 primary addresses that could possibly be connected to the IEEE-488 bus.

   c. Find out which, if any, device or devices are connected.

4. Send an identification query to each device for identification.

5. Initialize the instrument as follows:

   a. Clear the multimeter.

   b. Send the IEEE-488.2 Reset command to the meter.

6. Instruct the meter to measure volts alternating current (VAC) using auto-ranging (AUTO), to wait for a trigger from the Controller before starting a measurement (TRIGGER 2), and to assert the IEEE-488 Service Request signal line, SRQ, when the measurement has been completed and the meter is ready to send the result (*SRE 16).

7. For each measurement:

   a. Send the `TRIGGER` command to the multimeter. The command `"VAL1?"` instructs the meter to send the next triggered reading to its IEEE-488.2 output buffer.

   b. Wait until the DVM asserts Service Request (SRQ) to indicate that the measurement is ready to be read.

   c. Read the status byte to determine if the measured data is valid or if a fault condition exists. You can find out by checking the message available (MAV) bit, bit 4 in the status byte.

   d. If the data is valid, read 10 bytes from the DVM.

8. End the session.

The NI-488.2 software supports four interface boards. These boards are referenced by number from your application program. The reference number is zero (0) for the first board, one (1) for the second board, and so on. If you installed two or more boards in your computer, and you do not know which board is 0, which board is 1, and so on, run the configuration utility, `IBCONF`. `IBCONF` shows you the relationship between the board number and the base address of the board; thereby identifying the board by its base address. Refer to Chapter 2, *Installation and Configuration of the NI-488.2 Software*, in this manual for additional information about running and using `IBCONF`.

BASICA Example Program–NI-488.2 Routines

```
100  REM
110  REM You must merge this code with DECL.BAS.
120       dim instruments% (31)
130       dim result% (31)
140       READING$ = Space$(30)
150       boardindex% = 0
160  REM
170       CLS
180  REM
190  REM  Your interface board must be the Controller-In-
200  REM  Charge to perform the Find All Listeners protocol.
210  REM
220       Call SendIFC (boardindex%)
230       msg$ = "SendIFC Error"
240       If ibsta% and EERR THEN GOSUB 4000 : STOP
250  REM
260  REM  Create an array with all of the valid GPIB primary
270  REM  addresses.  This array will be given to the Find All
280  REM  Listeners protocol.
290  REM
300       For k% = 0 to 30
310          instruments% (k%) = k%
320       Next k%
330  REM
340       instruments%(31) = NOADDR%
350  REM
360  REM  Find all of the Listeners on the bus.
370  REM
380       Print "Finding all Listeners on the bus..."
390       limit% = 31
400  REM
410       CALL FindLstn (boardindex%, instruments%(0),
                          result%(0), limit%)
420       msg$ = "FindLstn Error"
430       If ibsta% and EERR then GOSUB 4000 : STOP
440  REM
450       num.Listeners% = ibcnt% - 1
460       Print "No. of instruments found = ", num.Listeners%
470  REM
480  REM  Now send the *IDN? command to each of the devices that
490  REM  you found.
500  REM
510  REM  The GPIB board is at address 0 by default.  Your
520  REM  GPIB board does not respond to *IDN?, so skip it.
530  REM
540       For k% = 1 to num.Listeners%
550           cmd$ = "*IDN?"
560           call Send(boardindex%, result%(k%), cmd$, NLend%)
570           msg$ = "Send Error"
580           If ibsta% and EERR then GOSUB 4000 : STOP
590  REM
```

```
600           call Receive(boardindex%,result%(k%),Reading$,
                             STOPend%)
610           msg$ = "Receive Error"
620           If ibsta% and EERR then GOSUB 4000 : STOP
630  REM
640           pad% = result%(k%) and &HFF
650           print "The instrument at address "; pad%; " is: ",
                       left$(Reading$, IBCNT%)
660           If left$(Reading$, 9)="FLUKE, 45" then GOTO 2000
670       Next k%
680  REM
690       Print "Did not find the Fluke!" : STOP


2000 REM  Device Found.
2010 REM
2020       Print "**** We found the Fluke 45 ****"
2030      fluke% = result%(k%)
2040 REM
2050 REM  Reset the Fluke.
2060 REM
2070      Call DevClear (boardindex%, fluke%)
2080      msg$ = "DevClear Error"
2090      If ibsta% and EERR then GOSUB 4000  :  STOP
2100 REM
2110      cmd$ = "*RST"
2120      Call Send(boardindex%, fluke%, cmd$, NLend%)
2130      msg$ = "Send *RST Error"
2140      If ibsta% and EERR then GOSUB 4000 : STOP
2150 REM
2160 REM  Set up for a test.  Allow the Fluke to assert
2170 REM  SRQ when it has a message to send.
2180 REM
2190      cmd$ = "VAC; AUTO; TRIGGER 2; *SRE 16"
2200      Call Send(boardindex%, fluke%, cmd$, NLend%)
2210      msg$ = "Send Setup Error"
2220      If ibsta% and EERR then GOSUB 4000 : STOP
2230 REM
2240      sum = 0
2250 REM
2260 REM  Trigger the Fluke.
2270 REM
2280      For m% = 1 to 10
2290          cmd$ = "*TRG; VAL1?"
2300          Call Send(boardindex%, fluke%, cmd$, NLend%)
2310          msg$ = "Send Trigger Error"
2320          If ibsta% and EERR then GOSUB 4000 : STOP
2330 REM
2340 REM  Wait for the Fluke to assert SRQ, meaning it is ready
2350 REM  with the measurement.
2360 REM
```

```
2370          Call WaitSRQ(boardindex%, SRQasserted%)
2380          msg$ = "WaitSRQ Error"
2390          If SRQasserted% = 0 then GOSUB 4000 : STOP
2400 REM
2410 REM  Read its status byte. Be sure that the Message
Available
2420 REM  bit is set.
2430 REM
2440          Call ReadStatusByte(boardindex%, fluke%,
                                  status%)
2450          msg$ = "ReadStatusByte Error"
2460          If ibsta% and EERR then GOSUB 4000 : STOP
2470 REM
2480          msg$ = "Improper Status Byte"
2490          If (status% and &H010) <> &H010 then GOSUB 4000
2500          If (status% and &H010) <> &H010 then Print "Status
                 byte: "; status% : STOP
2510 REM
2520 REM  Read the measurement.
2530 REM
2540          READING$ = Space$(30)
2550          Call Receive (boardindex%, fluke%, Reading$,
                             STOPend%)
2560          msg$ = "Receive Error"
2570          If ibsta% and EERR then GOSUB 4000 : STOP
2580 REM
2590          Reading$ = left$(Reading$, IBCNT%)
2600          Print "Reading: "; Reading$
2610 REM
2620          sum = sum + val(Reading$)
2630     Next m%
2640 REM
2650     Print "The average of the 10 readings is ", sum/10
2660 REM
2670 REM Call the ibonl function to disable the hardware and
2680 REM software.
2690 REM
2700     v% = 0 : CALL ibonl (boardindex%,v%) : STOP
2710   END

4000 REM Print Status Variables.
4010 REM
4020     locate 15,1
4030     Print msg$
4040 REM
4050     Print "ibsta=&H"; hex$(ibsta%); "< ";
4060     If ibsta% and EERR  then print " ERR";
4070     If ibsta% and TIMO then print " TIMO";
4080     If ibsta% and EEND then print " END";
4090     If ibsta% and SRQI then print " SRQI";
4100     If ibsta% and RQS  then print " RQS";
4110     If ibsta% and CMPL then print " CMPL";
4120     If ibsta% and LOK  then print " LOK";
```

```
4130     If ibsta% and RREM then print " REM";
4140     If ibsta% and CIC  then print " CIC";
4150     If ibsta% and AATN then print " ATN";
4160     If ibsta% and TACS then print " TACS";
4170     If ibsta% and LACS then print " LACS";
4180     If ibsta% and DTAS then print " DTAS";
4190     If ibsta% and DCAS then print " DCAS";
4200     Print ">"
4210 REM
4220     Print "iberr="; iberr%;
4230     If iberr% = EDVR then print " EDVR <DOS Error>"
4240     If iberr% = ECIC then print " ECIC <Not CIC>"
4250     If iberr% = ENOL then print " ENOL <No Listener>"
4260     If iberr% = EADR then print " EADR <Address error>"
4270     If iberr% = EARG then print " EARG <Invalid argument>"
4280     If iberr% = ESAC then print " ESAC <Not Sys Ctrlr>"
4290     If iberr% = EABO then print " EABO <Op. aborted>"
4300     If iberr% = ENEB then print " ENEB <No GPIB board>"
4310     If iberr% = EOIP then print " EOIP <Async I/O in prg>"
4320     If iberr% = ECAP then print " ECAP <No capability>"
4330     If iberr% = EFSO then print " EFSO <File sys. error>"
4340     If iberr% = EBUS then print " EBUS <Command error>"
4350     If iberr% = ESTB then print " ESTB <Status byte lost>"
4360     If iberr% = ESRQ then print " ESRQ <SRQ stuck on>"
4370     If iberr% = ETAB then print " ETAB <Table Overflow>"
4380 REM
4390     Print "ibcnt="; ibcnt% : RETURN
4400 REM
4410 REM Call the ibonl function to disable the hardware and
4420 REM software.
4430 REM
4440     ud% = 0 : v%= 0 : call ibonl(ud%, v%) : RETURN
```

QuickBASIC Example Program–NI-488.2 Routines

```
   REM $include: 'qbdecl.bas'

   declare sub gpiberr (msg$)

   dim instruments% (31)
   dim result% (30)
   dim READING as STRING * 30

   CLS

'  Your GPIB board must be the Controller-In-Charge to
'  perform the Find All Listeners protocol.

   Call SendIFC (0)
   If ibsta% and EERR then
        call gpiberr ("SendIFC Error")
        stop
   End If

'  Create an array with all of the valid GPIB primary addresses.
'  This array will be given to the Find All Listeners protocol.

   For k% = 0 to 30
        instruments% (k%) = k%
   Next k%
   instruments%(31) = NOADDR

'  Find all of the Listeners on the bus.

   Print "Finding all Listeners on the bus..."

   Call FindLstn (0, instruments%(), result%(), 31)
   If ibsta% and EERR then
        call gpiberr("FindLstn Error")
        stop
   End If
   num.Listeners% = ibcnt% - 1

   Print "No. of instruments found = ", num.Listeners%

'  Now send the *IDN? command to each of the devices that
'  you found.
'
'  The GPIB board is at address 0 by default.  Your GPIB board
'  does not respond to *IDN?, so skip it.
```

```
   For k% = 1 to num.Listeners%
        call Send(0, result%(k%), "*IDN?", NLend)
        if ibsta% and EERR then
            call gpiberr("Send Error")
            stop
        end if

        call Receive(0,result%(k%),Reading$, STOPend)
        if ibsta% and EERR then
            call gpiberr("Receive Error")
            stop
        end if

        pad% = result%(k%) and &HFF
        print "The instrument at address "; pad%; " is: ", _
              left$ (Reading$, IBCNT%)

        if left$(Reading$, 9)="FLUKE, 45" then
                fluke% = result%(k%)
                print "**** We found the Fluke 45 ****"
                goto found
        end if
   Next k%
   Print "Did not find the Fluke!"
   Stop

found:

'  Reset the Fluke.

   call DevClear (0, fluke%)
   If ibsta% and EERR then
        call gpiberr("DevClear Error")
        stop
   End If

   Call Send(0, fluke%, "*RST", NLend)
   If ibsta% and EERR then
        call gpiberr("Send *RST Error")
        stop
   End If

'  Set up for a test.  Allow the Fluke to assert SRQ when it
'  has a message to send.

   call Send(0, fluke%, "VAC; AUTO; TRIGGER 2; *SRE 16", NLend)
   If ibsta% and EERR then
        call gpiberr("Send Setup Error")
        stop
   End If

   sum = 0
   for m% = 1 to 10
```

```
        '  Trigger the Fluke.

           call Send(0, fluke%, "*TRG; VAL1?", NLend)
           if ibsta% and EERR then
                call gpiberr("Send Trigger Error")
                stop
           end if

        '  Wait for the Fluke to assert SRQ, meaning it is ready
        '  with the measurement.

           call WaitSRQ(0, SRQasserted%)
           if SRQasserted% = 0 then
                call gpiberr("WaitSRQ Error")
                stop
           end if

        '  Read its status byte.  Make sure that the MAV
        '  (Message Available) bit is set.

           call ReadStatusByte (0, fluke%, status%)
           if ibsta% and EERR then
                call gpiberr("ReadStatusByte Error")
                stop
           end if

           if (status% and &H010) <> &H010 then
                call gpiberr("Improper Status Byte")
                print "Status Byte: "; status%
                stop
           end if

        '  Read the measurement.

           call Receive (0, fluke%, Reading$, STOPend)
           if ibsta% and EERR then
                call gpiberr("Receive Error")
                stop
           end if

          Reading$ = left$(Reading$, IBCNT%)
          print "Reading: "; Reading$

          sum = sum + val (Reading$)
     Next m%

     Print "The average of the 10 readings is ", sum/10

'  Call the ibonl function to disable the hardware and software.

   Call ibonl(0,0)

End
```

```
'  This routine prints the result of the status variables.

sub gpiberr(msg$) static

locate 15,1
print msg$

   print "ibsta=&H"; hex$(ibsta%); "< ";
   if ibsta% and EERR then print " ERR";
   if ibsta% and TIMO then print " TIMO";
   if ibsta% and EEND then print " END";
   if ibsta% and SRQI then print " SRQI";
   if ibsta% and RQS  then print " RQS";
   if ibsta% and CMPL then print " CMPL";
   if ibsta% and LOK  then print " LOK";
   if ibsta% and RREM then print " REM";
   if ibsta% and CIC  then print " CIC";
   if ibsta% and AATN then print " ATN";
   if ibsta% and TACS then print " TACS";
   if ibsta% and LACS then print " LACS";
   if ibsta% and DTAS then print " DTAS";
   if ibsta% and DCAS then print " DCAS";
   print ">"

   print "iberr="; iberr%;
   if iberr% = EDVR then print " EDVR <DOS Error>"
   if iberr% = ECIC then print " ECIC <Not CIC>"
   if iberr% = ENOL then print " ENOL <No Listener>"
   if iberr% = EADR then print " EADR <Address error>"
   if iberr% = EARG then print " EARG <Invalid argument>"
   if iberr% = ESAC then print " ESAC <Not Sys Ctrlr>"
   if iberr% = EABO then print " EABO <Op. aborted>"
   if iberr% = ENEB then print " ENEB <No GPIB board>"
   if iberr% = EOIP then print " EOIP <Async I/O in prg>"
   if iberr% = ECAP then print " ECAP <No capability>"
   if iberr% = EFSO then print " EFSO <File sys. error>"
   if iberr% = EBUS then print " EBUS <Command error>"
   if iberr% = ESTB then print " ESTB <Status byte lost>"
   if iberr% = ESRQ then print " ESRQ <SRQ stuck on>"
   if iberr% = ETAB then print " ETAB <Table Overflow>"

   print "ibcnt="; ibcnt%

'  Call the ibonl function to disable the hardware and software.

   call ibonl(0, 0)

end sub
```

Microsoft BASIC Example Program–NI-488.2 Routines

```
   REM $include: 'mbdecl.bas'

   DECLARE SUB gpiberr(msg$)

   DIM instruments% (31)
   DIM result% (30)
   DIM READING as STRING * 30

   CLS

'  Your GPIB board must be the Controller-In-Charge to
'  perform the Find All Listeners protocol.

   Call SendIFC(0)
   If ibsta% and EERR then
        call gpiberr ("SendIFC Error")
        stop
   End If

'  Create an array with all of the valid GPIB primary addresses.
'  This array will be given to the Find All Listeners protocol.

   For k% = 0 to 30
        instruments% (k%) = k%
   Next k%
   instruments%(31) = NOADDR

'  Find all of the Listeners on the bus.

   Print "Finding all Listeners on the bus..."

   Call FindLstn (0, instruments%(), result%(), 31)
   If ibsta% and EERR then
        call gpiberr("FindLstn Error")
        stop
   End If
   num.Listeners% = ibcnt% - 1

   Print "No. of instruments found = ", num.Listeners%

'  Now send the *IDN? command to each of the devices that you
'  found.
'
'  The GPIB board is at address 0 by default.  Your GPIB board
'  does not respond to *IDN?, so skip it.
```

```
   For k% = 1 to num.Listeners%
        call Send(0, result%(k%), "*IDN?", NLend)
        if ibsta% and EERR then
            call gpiberr("Send Error")
            stop
        end if
        call Receive(0,result%(k%),Reading$, STOPend)
        if ibsta% and EERR then
            call gpiberr("Receive Error")
            stop
        end if

        pad% = result%(k%) and &HFF
        print "The instrument at address "; pad%; " is: ", left$
              (Reading$, IBCNT%)

        if left$(Reading$, 9)="FLUKE, 45" then
                fluke% = result%(k%)
                print "**** We found the Fluke 45 ****"
                goto found
        end if
   Next k%
   Print "Did not find the Fluke!"
   Stop

found:

'  Reset the Fluke.

   Call DevClear (0, fluke%)
   If ibsta% and EERR then
        call gpiberr("DevClear Error")
        stop
   End If

   Call Send(0, fluke%, "*RST", NLend)
   If ibsta% and EERR then
        call gpiberr("Send *RST Error")
        stop
   End If

'  Set up for a test.  Allow the Fluke to assert SRQ when it
'  has a message to send.

   Call Send(0, fluke%, "VAC; AUTO; TRIGGER 2; *SRE 16", NLend)
   If ibsta% and EERR then
        call gpiberr("Send Setup Error")
        stop
   End If
   sum = 0
   For m% = 1 to 10

        '  Trigger the Fluke.

        call Send(0, fluke%, "*TRG; VAL1?", NLend)
        if ibsta% and EERR then
             call gpiberr("Send Trigger Error")
             stop
        end if

        '  Wait for the Fluke to assert SRQ, meaning it is ready
        '  with the measurement.

        call WaitSRQ(0, SRQasserted%)
        if SRQasserted% = 0 then
             call gpiberr("WaitSRQ Error")
             stop
        end if

        '  Read its status byte.  Make sure that the MAV
        '  (Message Available) bit is set.

        call ReadStatusByte (0, fluke%, status%)
        if ibsta% and EERR then
             call gpiberr("ReadStatusByte Error")
             stop
        end if

        if (status% and &H010) <> &H010 then
             call gpiberr("Improper Status Byte")
             print "Status Byte: "; status%
             stop
        end if

        '  Read the measurement.

        call Receive (0, fluke%, Reading$, STOPend)
        if ibsta% and EERR then
             call gpiberr("Receive Error")
             stop
        end if

        Reading$ = left$(Reading$, IBCNT%)
        print "Reading: "; Reading$

        sum = sum + val (Reading$)
   Next m%

   Print "The average of the 10 readings is ", sum/10

'  Call the ibonl function to disable the hardware and software.
```

```
   Call ibonl(0,0)
End


'  This routine prints the result of the status variables.

Sub gpiberr(msg$) static

   Locate 15,1
   Print msg$

   Print "ibsta=&H"; hex$(ibsta%); "< ";
   If ibsta% and EERR then print " ERR";
   If ibsta% and TIMO then print " TIMO";
   If ibsta% and EEND then print " END";
   If ibsta% and SRQI then print " SRQI";
   If ibsta% and RQS then print " RQS";
   If ibsta% and CMPL then print " CMPL";
   If ibsta% and LOK then print " LOK";
   If ibsta% and RREM then print " REM";
   If ibsta% and CIC then print " CIC";
   If ibsta% and AATN then print " ATN";
   If ibsta% and TACS then print " TACS";
   If ibsta% and LACS then print " LACS";
   If ibsta% and DTAS then print " DTAS";
   If ibsta% and DCAS then print " DCAS";
   Print ">"

   Print "iberr="; iberr%;
   If iberr% = EDVR then print " EDVR <DOS Error>"
   If iberr% = ECIC then print " ECIC <Not CIC>"
   If iberr% = ENOL then print " ENOL <No Listener>"
   If iberr% = EADR then print " EADR <Address error>"
   If iberr% = EARG then print " EARG <Invalid argument>"
   If iberr% = ESAC then print " ESAC <Not Sys Ctrlr>"
   If iberr% = EABO then print " EABO <Op. aborted>"
   If iberr% = ENEB then print " ENEB <No GPIB board>"
   If iberr% = EOIP then print " EOIP <Async I/O in prg>"
   If iberr% = ECAP then print " ECAP <No capability>"
   If iberr% = EFSO then print " EFSO <File sys. error>"
   If iberr% = EBUS then print " EBUS <Command error>"
   If iberr% = ESTB then print " ESTB <Status byte lost>"
   If iberr% = ESRQ then print " ESRQ <SRQ stuck on>"
   If iberr% = ETAB then print " ETAB <Table Overflow>"

   Print "ibcnt="; ibcnt%

'  Call the ibonl function to disable the hardware and software.

   CALL IBONL(0, 0)

End Sub
```

C Example Program–NI-488.2 Routines

```
/*
 * Link this program with mcib.obj
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "decl.h"

/* Position of the Message Available bit. */

#define  MAVbit   0x10

   unsigned char   buffer[101];
   short           loop, m, num_Listeners, SRQasserted;
   double          sum;
   unsigned short  instruments[32], result[31], fluke,
statusByte;

void gpiberr(char *msg);             /* gpib error function
                                                          */


main() {

   system("cls");

/* Your GPIB board must be the Controller-In-Charge to perform
 * the Find All Listeners protocol.
 */
   SendIFC(0);
   if (ibsta & ERR) {
        gpiberr ("SendIFC Error");
        exit(1);
   }


/* Create an array with all of the valid GPIB primary addresses.
 * This array will be given to the Find All Listeners protocol.
 */
   for (loop = 0; loop <= 30; loop++) {
      instruments[loop] = loop;
   }
   instruments[31] = NOADDR;    /* Mark the end of the array.
                                                          */


/* Find all of the Listeners on the bus.
 */
   printf("Finding all Listeners on the bus...\n");
   FindLstn(0, instruments, result, 31);
   if (ibsta & ERR) {
        gpiberr("FindLstn Error");
        exit(1);
   }

   num_Listeners = ibcnt - 1;

   printf("Number of instruments found = %d\n", num_Listeners);

/* Now send the *IDN? command to each of the devices that you
 * found.
 *
 * The GPIB board is at address 0 by default.  Your GPIB board
 * does not respond to *IDN?, so skip it.
 */
   for (loop = 1; loop <= num_Listeners; loop++) {
      Send(0, result[loop], "*IDN?", 5L, NLend);
      if (ibsta & ERR) {
           gpiberr("Send Error");
           exit(1);
      }

      Receive(0, result[loop], buffer, 10L, STOPend);
      if (ibsta & ERR) {
           gpiberr("Receive Error");
           exit(1);
      }

      buffer[ibcnt] = '\0';
      printf("The instrument at address %d is a %s\n",
GetPAD(result[loop]),
                buffer);

      if (strncmp(buffer, "FLUKE, 45", 9) == 0) {
         fluke = result[loop];
         printf("**** We found the Fluke ****\n");
         break;
      }
   }

   if (loop > num_Listeners) {
        printf("Did not find the Fluke!\n");
        exit(1);
   }

/* Reset the Fluke.
 */
   DevClear(0, fluke);
   if (ibsta & ERR) {
        gpiberr("DevClear Error");
        exit(1);
   }
   Send(0, fluke, "*RST", 4L, NLend);
   if (ibsta & ERR) {
        gpiberr("Send *RST Error");
        exit(1);
   }

/* Set up for a test.  Allow the Fluke to assert SRQ when it
 * has a message to send.
 */
   Send(0, fluke, "VAC; AUTO; TRIGGER 2; *SRE 16", 29L, NLend);
   if (ibsta & ERR) {
      gpiberr("Send Setup Error");
      exit(1);
   }

   sum = 0.0;
   for (m=0; m < 10 ; m++) {

        /*  Trigger the Fluke.
         */
         Send(0, fluke, "*TRG; VAL1?", 11L, NLend);
         if (ibsta & ERR) {
            gpiberr("Send Trigger Error");
            exit(1);
         }

        /*  Wait for the Fluke to assert SRQ, meaning it is
ready
         *  with the measurement.
         */
        WaitSRQ(0, &SRQasserted);
        if (!SRQasserted) {
            printf("SRQ is not asserted.  The Fluke is not ");
            printf("ready.\n");
            exit(1);
        }

        /*  Read its status byte.  Be sure that the MAV (Message
         *  Available) bit is set.
         */
         ReadStatusByte(0, fluke, &statusByte);
         if (ibsta & ERR) {
            gpiberr("ReadStatusByte Error");
            exit(1);
         }

         if (!(statusByte & MAVbit)) {
             gpiberr("Improper Status Byte");
             printf(" Status Byte = 0x%x\n", statusByte);
             exit(1);
         }
        /*  Read the measurement.
         */
         Receive(0, fluke, buffer, 10L, STOPend);
         if (ibsta & ERR) {
            gpiberr("Receive Error");
            exit(1);
         }

         buffer[ibcnt] = '\0';

         printf("Reading :  %s\n", buffer);
         sum = sum + atof(buffer);
   }

   printf("   The average of the 10 readings is :  ");
   printf("   %f\n", sum/10);

/* Call the ibonl function to disable the hardware and software.
*/

   ibonl (0,0);

}


void gpiberr(char *msg) {

   printf ("%s\n", msg);

   printf ( "ibsta=&H%x ", ibsta, "< ");
   if (ibsta & ERR ) printf (" ERR");
   if (ibsta & TIMO) printf (" TIMO");
   if (ibsta & END ) printf (" END");
   if (ibsta & SRQI) printf (" SRQI");
   if (ibsta & RQS ) printf (" RQS");
   if (ibsta & CMPL) printf (" CMPL");
   if (ibsta & LOK ) printf (" LOK");
   if (ibsta & REM ) printf (" REM");
   if (ibsta & CIC ) printf (" CIC");
   if (ibsta & ATN ) printf (" ATN");
   if (ibsta & TACS) printf (" TACS");
   if (ibsta & LACS) printf (" LACS");
   if (ibsta & DTAS) printf (" DTAS");
   if (ibsta & DCAS) printf (" DCAS");
   printf (">\n");

   printf ("iberr= %d", iberr);
   if (iberr == EDVR) printf (" EDVR <DOS Error>\n");
   if (iberr == ECIC) printf (" ECIC <Not CIC>\n");
   if (iberr == ENOL) printf (" ENOL <No Listener>\n");
   if (iberr == EADR) printf (" EADR <Address error>\n");
   if (iberr == EARG) printf (" EARG <Invalid argument>\n");
   if (iberr == ESAC) printf (" ESAC <Not Sys Ctrlr>\n");
   if (iberr == EABO) printf (" EABO <Op. aborted>\n");
   if (iberr == ENEB) printf (" ENEB <No GPIB board>\n");
   if (iberr == EOIP) printf (" EOIP <Async I/O in prg>\n");
   if (iberr == ECAP) printf (" ECAP <No capability>\n");
   if (iberr == EFSO) printf (" EFSO <File sys. error>\n");
   if (iberr == EBUS) printf (" EBUS <Command error>\n");
   if (iberr == ESTB) printf (" ESTB <Status byte lost>\n");
   if (iberr == ESRQ) printf (" ESRQ <SRQ stuck on>\n");
   if (iberr == ETAB) printf (" ETAB <Table Overflow>\n");

   printf ("ibcnt= %d\n", ibcntl);
   printf ("\n");

/* Call the ibonl function to disable the hardware and software.
*/

   ibonl (0,0);

   }
```

Chapter 5
NI-488 Software Characteristics and Functions

This chapter contains a discussion of the important characteristics of the NI-488 functions available in the MS-DOS driver that are common to all programming languages. It also contains a detailed description of each NI-488 function with example programs. The description are listed alphabetically for easy reference.

If you order a language interface for languages other than BASICA, QuickBASIC, BASIC, or C, you receive a separate manual.

General Programming Information

The following facilities or operations are common to all programming languages:

- Status Word (`ibsta`)
- Error Codes (`iberr`)
- Count Variables (`ibcnt` and `ibcntl`)
- Read and Write Termination
- Device Functions
- Automatic Serial Polling

You should understand these topics thoroughly to take full advantage of the NI-488.2 for MS-DOS driver's capabilities. Refer to Chapter 3, *Understanding the NI-488.2 Software*, for information on the status word (`ibsta`), the error variable (`iberr`), and the count variables (`ibcnt` and `ibcntl`). These variables are updated with each function call to reflect the status of the device or board just accessed.

Device Functions

Device functions are those functions in which the unit descriptor identifies a device rather than an interface board (refer to Chapter 3 for a discussion of boards and devices). There are some activities common to all device functions that you should understand thoroughly.

In a single-board configuration, there is only one GPIB interface board in use. When the first device function of a program is executed, the software initializes the IEEE-488 bus by sending the Interface Clear (IFC) single-line interface message. The Remote Enable (REN) line on the IEEE-488 bus is also asserted, and the Local Lockout (LLO) multiline interface message is sent to all devices on the IEEE-488 bus to place them in a lockout state. Furthermore, the device may be addressed to listen and then unaddressed before certain functions are executed. This step ensures that the device is in remote program mode.

In a multiboard configuration, more than one GPIB board is in use. The process is the same as in the preceding description, with the exception that each IEEE-488 bus is initialized by its associated GPIB interface board when the first device on that IEEE-488 bus is accessed by a device function.

The preceding descriptions assume that the GPIB board is the System Controller of its IEEE-488 bus, which is the usual configuration. As long as the GPIB interface board is the System Controller, it can become the Controller-In-Charge (CIC). If the GPIB board is not CIC or is unable to become CIC, it is not able to execute device functions, and any call to a device function that uses the IEEE-488 bus returns an ECIC error.

The software can also be configured to request control from the current CIC by using the configuration utility, `IBCONF`, or the configuration function, `ibconfig`. If the request CIC configuration option is enabled, and the GPIB interface board is not CIC when a device function is called, the board requests service from the current CIC by asserting the SRQ line and passing the status response byte (hex 42) when serial polled by the CIC.

The software waits (using the current timeout value of the board) for control to be passed to the GPIB interface board. If the board fails to become CIC before the timeout period elapses, an ECIC error is returned. This error might occur, for example, if the GPIB interface board is assumed to be the System Controller, but was not configured as such during software installation.

Automatic Serial Polling

Automatic Serial Polling is a feature of the software that is intended to relieve you from the burden of sorting out occurrences of SRQ and status bytes of a device. Automatic Serial Polling (or *Autopolling*) can be enabled by using the configuration utility, `IBCONF`, or by using the configuration function, `ibconfig`. If Autopolling is enabled, the software automatically conducts serial polls when SRQ is asserted.

As part of the Autopoll procedure, the software stores each positive serial poll response–that is, those responses that have the RQS or hex 40 bit set in the device status byte, in a queue associated with each device. Queues are necessary because some devices can send multiple positive status bytes back-to-back. When a positive response from a device is received, the RQS bit of its status word (`ibsta`) is set. The polling continues until SRQ is unasserted or an error condition is detected.

If the software cannot locate the device requesting service (no known device responds positively to the poll) or if SRQ becomes stuck on (because of a faulty instrument or cable), a GPIB system error exists that interferes with the proper evaluation of the RQS bit in the status words of devices. The error ESRQ is reported to you if and when you issue an `ibwait` call with the RQS bit included in the wait mask. Should the error condition correct itself, you should notice it by calling `ibwait` with the RQS bit set in the mask, where the ESRQ error is not reported. Aside from the difficulty caused by ESRQ in waiting for RQS, the error has no detrimental effects on other GPIB operations.

If the serial poll function `ibrsp` is called and one or more responses have been received previously via the automatic serial poll feature, the first queued response is returned by the `ibrsp` function. Other responses are read in first-in-first-out (FIFO) fashion. If the RQS bit of the status word is not set when `ibrsp` is called, the function conducts a serial poll and returns whatever response is received.

If your application requires that requests for service be noticed, you should examine the RQS bit in the status word and call the `ibrsp` function to examine the status byte whenever it appears. It is possible for a serial poll response queue of a device to get clogged with old status bytes when you neglect to call `ibrsp` to empty the queue. Error condition ESTB is returned only by `ibrsp` when it is necessary to report that status bytes have been discarded due to a full queue. If your application has no interest in SRQ or status bytes, you can ignore the occurrence of the automatic polls.

Note: If the RQS bit of the device status word is still set after `ibrsp` is called, the response byte queue has at least one more response byte remaining. `ibrsp` should be called until RQS is cleared to gather all stored response bytes and to guard against queue overflow.

Compatibility

Autopolling is incompatible with the "ON SRQ" feature in some language interfaces. That is, either the user or the driver can be in charge of handling SRQ, but not both. You should disable Autopolling if you intend to use "ON SRQ". Refer to Chapter 3, *Understanding the NI-488.2 Software*, for a description of the NI-488.2 driver's "ON SRQ" feature.

Note: Some mouse drivers conflict with the "ON SRQ" feature. You may need to disable your mouse driver before using "ON SRQ."

Board functions are incompatible with Autopolling. Refer to Chapter 3 for a description of board calls. If Autopolling were to occur after a board call, it could, in some cases, undo the effect of the call. For example, if SRQ were asserted immediately after an `ibcmd` call had been made to address a device, Autopoll addressing of other devices for serial polls would destroy your intentions of addressing the device with which you must communicate. For this reason, the driver disables Autopolling whenever a board call is made, and turns it back on once the application program makes a device call (actually, Autopolling is turned on at the end of the subsequent device call).

Internal Driver Operation

If Autopolling is enabled, whenever SRQ is asserted on the IEEE-488 bus, all online devices–that is, all devices that have been opened using `ibfind` or `ibdev`, connected to the GPIB interface board that detected the request are serial polled by the driver until one of the following events occur:

- SRQ unasserts.
- All online devices have been polled since the last positive response and SRQ still remains asserted. Once this has occurred, a "stuck SRQ" state is in effect inside the driver. If this state is reached during an `ibwait` for RQS, the ESRQ error is reported for that `ibwait` call and the "stuck SRQ" state is terminated. If the "stuck SRQ" state is reached at some other time, further polls are not attempted until an `ibwait` for RQS is made, at which time the "stuck SRQ" state is terminated and a new set of serial polls is attempted. In addition, the "stuck SRQ" state is immediately terminated whenever SRQ is found to be unasserted.

Hardware Interrupts

If the interrupts of the GPIB interface board are disabled by using `IBCONF` or the `ibconfig` function, the driver detects SRQ only during calls to the driver, and Autopolling can occur only at the following events:

- During a device `ibwait` for RQS
- Immediately after a device function is completed and is about to return to the application program

If hardware interrupts are enabled, in addition to the events just noted, the driver can respond to SRQI interrupts and perform Autopolling even if the driver is not performing a function. However, even during these intervals when Autopolling can occur, the driver will not conduct an Autopoll if any of the following conditions exist:

- Immediately after a board call or NI-488.2 call. Autopolling is re-instated after a subsequent device call.
- While GPIB I/O is in progress. In particular, during asynchronous GPIB I/O, Autopolling will not occur until all driver operations are complete–that is, until the asynchronous I/O has completed.

- When the "stuck SRQ" condition exists.

- Autopolling is disabled by `IBCONF` or by `ibconfig`. However, during an `ibwait` for RQS, the device to which the call was directed is serial polled. If SRQ becomes asserted during this call and a negative poll response is received from the device in question, the "stuck SRQ" condition is returned.

BASICA/QuickBASIC/BASIC/C NI-488 I/O Calls and Functions

The most commonly needed I/O calls are `ibrd` and `ibwrt`. In BASICA, these functions read and write from a character string that may be up to 255 bytes long.

In addition, integer I/O calls (`ibrdi` and `ibwrti`) are for data strings longer than 255 bytes, or for users who need to perform arithmetic operations on the data and want to avoid the overhead of converting the character bytes of `ibrd` and `ibwrt` into integer format and back again. `ibrdi` and `ibwrti` are passed data in the form of an integer array so that more than 255 bytes can be transferred. Internally, the `ibwrti` function sends each integer to the GPIB in low-byte, high-byte order. The `ibrdi` function reads a series of data bytes from the GPIB and stores them into the integer array in low-byte, high-byte order.

In addition to `ibrdi` and `ibwrti`, the asynchronous functions `ibrdia` and `ibwrtia` perform asynchronous integer reads and writes.

The first argument of all calls and functions except `ibdev`, `ibfind`, and `ibtrap` is the integer variable `ud`. This serves as a general unit descriptor to show the format of the calls. In practice, `ud` refers to the board or device to which the command is directed. Refer to the *IBFIND* function description in this chapter and to Chapter 3, *Understanding the NI-488.2 Software*, to determine the type of unit descriptor to use.

The functions are listed alphabetically by function name in this chapter. Tables 5-1, 5-2, and 5-3 list the functions and a description of BASICA, QuickBASIC (version 4.0 and later)/BASIC (version 7.0), and C NI-488 functions, respectively.

Table 5-1. BASICA NI-488 Functions

Call Syntax	**Description**
`ibbna (ud%,bname$)`	Change access board of device
`ibcac (ud%,v%)`	Become Active Controller
`ibclr (ud%)`	Clear specified device
`ibcmd (ud%,cmd$)`	Send commands from string
`ibcmda (ud%,cmd$)`	Send commands asynch. from string
`ibconfig (ud%,option%,value%)`	Configure the software
`ibdev (board.index%,pad%, sad%,tmo%,eot%,eos%,ud%)`	Open and initialize an unused device when the device name is unknown
`ibdma (ud%,v%)`	Enable/disable DMA
`ibeos (ud%,v%)`	Change/disable EOS mode
`ibeot (ud%,v%)`	Enable/disable END message
`ibevent (ud%,event%)`	Return the next event
`ibfind (udname$,ud%)`	Open device and return unit descriptor
`ibgts (ud%,v%)`	Go from Active Controller to standby
`ibist (ud%,v%)`	Set/clear ist
`iblines (board.index%,lines%)`	Get status of GPIB lines
`ibln (ud%,pad%,sad%,listen%)`	Check for presence of device on bus.
`ibloc (ud%)`	Go to local
`ibonl (ud%,v%)`	Place device online/offline
`ibpad (ud%,v%)`	Change primary address
`ibpct (ud%)`	Pass control
`ibppc (ud%,v%)`	Parallel poll configure
`ibrd (ud%,rd$)`	Read data to string
`ibrda (ud%,rd$)`	Read data asynchronously to string
`ibrdf (ud%,flname$)`	Read data to file

(continues)

Table 5-1. BASICA NI-488 Functions (continued)

Call Syntax	Description
`ibrdi (ud%,iarr%(0),cnt%)`	Read data to integer array
`ibrdia (ud%,iarr%(0),cnt%)`	Read data asynch. to integer array
`ibrpp (ud%,ppr%)`	Conduct a parallel poll
`ibrsc (ud%,v%)`	Request/release system control
`ibrsp (ud%,spr%)`	Return serial poll byte
`ibrsv (ud%,v%)`	Request service
`ibsad (ud%,v%)`	Change secondary address
`ibsic (ud%)`	Send interface clear
`ibsre (ud%,v%)`	Set/clear remote enable line
`ibstop (ud%)`	Abort asynchronous operation
`ibtmo (ud%,v%)`	Change/disable time limit
`ibtrap (mask%,v%)`	Configure applications monitor
`ibtrg (ud%)`	Trigger selected device
`ibwait (ud%,mask%)`	Wait for selected event
`ibwrt (ud%,wrt$)`	Write data from string
`ibwrta (ud%,wrt$)`	Write data asynchronously from string
`ibwrtf (ud%,flname$)`	Write data from file
`ibwrti (ud%,iarr%(0),cnt%)`	Write data from integer array
`ibwrtia (ud%,iarr%(0),cnt%)`	Write data asynch. from integer array

Table 5-2. QuickBASIC/BASIC NI-488 Functions

Call Syntax	Description
`ibbna (ud%,bname$)`	Change access board of device
`ibcac (ud%,v%)`	Become Active Controller
`ibclr (ud%)`	Clear specified device
`ibcmd (ud%,cmd$)`	Send commands from string
`ibcmda (ud%,cmd$)`	Send commands asynch. from string
`ibconfig (ud%,option%,value%)`	Configure the software
`ibdev (board.index%,pad%,`	Open and initialize an unused device
`sad%,tmo%,eot%,eos%,ud%)`	when the device name is unknown
`ibdma (ud%,v%)`	Enable/disable DMA
`ibeos (ud%,v%)`	Change/disable EOS mode
`ibeot (ud%,v%)`	Enable/disable END message
`ibevent (ud%,event%)`	Return the next event
`ibfind (brdname$,ud%)`	Open device and return unit descriptor
`ibgts (ud%,v%)`	Go from Active Controller to standby
`ibist (ud%,v%)`	Set/clear ist
`iblines (board.index%,lines%)`	Get status of GPIB lines
`ibln (ud%,pad%,sad%,listen%)`	Check for presence of device on bus.
`ibloc (ud%)`	Go to local
`ibonl (ud%,v%)`	Place device online/offline
`ibpad (ud%,v%)`	Change primary address
`ibpct (ud%)`	Pass control
`ibppc (ud%,v%)`	Parallel poll configure
`ibrd (ud%,rd$)`	Read data to string
`ibrda (ud%,rd$)`	Read data asynchronously to string
`ibrdf (ud%,flname$)`	Read data to file

(continues)

Table 5-2. QuickBASIC/BASIC NI-488 Functions (continued)

Call Syntax	Description
`ibrdi (ud%,iarr%(),cnt%)`	Read data to integer array
`ibrdia (ud%,iarr%(),cnt%)`	Read data asynch. to integer array
`ibrpp (ud%,ppr%)`	Conduct a parallel poll
`ibrsc (ud%,v%)`	Request/release system control
`ibrsp (ud%,spr%)`	Return serial poll byte
`ibrsv (ud%,v%)`	Request service
`ibsad (ud%,v%)`	Change secondary address
`ibsic (ud%)`	Send interface clear
`ibsre (ud%,v%)`	Set/clear remote enable line
`ibstop (ud%)`	Abort asynchronous operation
`ibtmo (ud%,v%)`	Change/disable time limit
`ibtrap (mask%,v%)`	Configure applications monitor
`ibtrg (ud%)`	Trigger selected device
`ibwait (ud%,mask%)`	Wait for selected event
`ibwrt (ud%,wrt$)`	Write data from string
`ibwrta (ud%,wrt$)`	Write data asynchronously from string
`ibwrtf (ud%,flname$)`	Write data from file
`ibwrti (ud%,iarr%(),cnt%)`	Write data from integer array
`ibwrtia (ud%,iarr%(),cnt%)`	Write data asynch. from integer array

Table 5-3. C Language NI-488 Functions

Call Syntax	Description
`ibbna (ud,bname)`	Change access board of device
`ibcac (ud,v)`	Become Active Controller
`ibclr (ud)`	Clear specified device
`ibcmd (ud,cmd,cnt)`	Send commands from string
`ibcmda (ud,cmd,cnt)`	Send commands asynch. from string
`ibconfig (ud,option,value)`	Configure the software
`ud = ibdev (bd_index,`	Open and initialize an unused device
`pad,sad,tmo,eot,eos)`	when the device name is unknown
`ibdma (ud,v)`	Enable/disable DMA
`ibeos (ud,v)`	Change/disable EOS mode (write)
`ibeot (ud,v)`	Enable/disable END message
`ibevent (ud,&event)`	Return the next event
`ud = ibfind (udname)`	Open device and return unit descriptor
`ibgts (ud,v)`	Go from Active Controller to Standby
`ibist (ud,v)`	Set/clear ind. status bit for Parallel Polls
`iblines (board,lines)`	Get status of GPIB lines
`ibln (pad,sad,listen)`	Check for presence of device on bus.
`ibloc (ud)`	Go to Local
`ibonl (ud,v)`	Place device or board online/offline
`ibpad (ud,v)`	Change Primary Address
`ibpct (ud)`	Pass Control
`ibppc (ud,v)`	Parallel Poll Configure
`ibrd (ud,rd,cnt)`	Read data to string
`ibrda (ud,rd,cnt)`	Read data asynchronously to string

(continues)

Table 5-3. C Language NI-488 Functions (continued)

Call Syntax	Description
`ibrdf (ud,flname)`	Read data to file
`ibrpp (ud,&ppr)`	Conduct a Parallel Poll
`ibrsc (ud,v)`	Request/release System Control
`ibrsp (ud,&spr)`	Return serial poll byte
`ibrsv (ud,v)`	Request service, set/change serial poll
`ibsad (ud,v)`	Change/disable Secondary Address
`ibsic (ud)`	Send Interface Clear for 100 µsec
`ibsre (ud,v)`	Set/clear Remote Enable line
`ibsrq (func)`	Register an SRQ "interrupt routine"
`ibstop (ud)`	Abort asynchronous operation
`ibtmo (ud,v)`	Change/disable time limit
`ibtrap(mask,mode)`	Configure Applications Monitor
`ibtrg (ud)`	Trigger selected device
`ibwait (ud,mask)`	Wait for selected event
`ibwrt (ud,wrt,cnt)`	Write data from string
`ibwrta (ud,wrt,cnt)`	Write data asynchronously from string
`ibwrtf (ud,flname)`	Write data from file

NI-488 IL Functions

Because QuickBASIC (versions 4.0 and later) and BASIC (version 7.0 and later) support functions in addition to subroutines, the language interface has been expanded to include function versions of the existing NI-488 calls. Here are some important points to understand:

- All NI-488 subroutines (`ibrd`, `ibwrt`, …) are available via the `CALL` statement.
- The names of the new functions are identical to the existing subroutine names, except that the second letter of each name has been changed from `b` to `l`. For example, the subroutine `ibsic` is also available as the function `ilsic`.
- GPIB subroutine and function calls may be freely mixed throughout a QuickBASIC 4.x and BASIC 7.x program.
- The include file `QBDECL.BAS` (`MBDECL.BAS` for BASIC) contains a complete list of both subroutine and function declarations, complete with parameter list specifications to aid in type checking at compile time. You must include this file in all application programs using NI-488 function calls.
- In general, the functions behave identically to the subroutines with the few exceptions noted in the following paragraph. The description of each subroutine found in this manual can be applied to the functions, except for the syntax-specific information.

There are a few differences between the existing subroutines and the new functions:

- `ilfind` returns a descriptor associated with the specified board or device. Use this value in all subsequent functions that access that device. Normal usage would resemble the following:

```
ud% = ilfind ("GPIB0")
```

- `ildev` opens and initializes an unused device when the device name is unknown. Normal usage would resemble the following:

```
ud% = ildev (0, 6, &H67, 13, 7, 0)
```

- `ilcmd`, `ilcmda`, `ilrd`, `ilrda`, `ilwrt`, and `ilwrta` require a third parameter which specifies the number of bytes to transfer. The function syntax is as follows:

```
sta% = ilcmd (ud%, cmd$, cnt%)
sta% = ilcmda (ud%, cmd$, cnt%)
sta% = ilrd (ud%, rd$, cnt%)
sta% = ilrda (ud%, rd$, cnt%)
sta% = ilwrt (ud%, wrt$, cnt%)
sta% = ilwrta (ud%, wrt$, cnt%)
```

- All functions, except `ilfind` and `ildev`, return the value of `ibsta`. This permits the following construct:

```
IF ilrd (ud%, rd%, cnt%) AND EERR THEN CALL GPIBERROR
```

Writing an NI-488 Program

The following paragraphs demonstrate how to use the NI-488 functions. An example program written with QuickBASIC is developed step-by-step. The example program uses device functions because they are the simplest functions and can be used for most applications. This program configures a digital multimeter (meter), reads back 10 voltage measurements, and computes the average of these measurements. For this example, assume that error detection and reporting is handled by the Applications Monitor. Refer to Chapter 7, *Applications Monitor*, for more information about APPMON.

At the end of this example program, an equivalent example program that uses board functions is shown. At the end of this chapter, device function programs are shown in BASIC, BASICA, and C languages.

Step 1. Initializing the System

The first step in writing a QuickBASIC program is to load in the definitions of the NI-488 functions from a file that is provided on your distribution diskette.

```
REM $INCLUDE: 'QBDECL.BAS'
```

The first NI-488 function call is to `ibfind` or `ibdev` to open a device.

```
CALL ibdev (0, 1, 0, 12, 1, 0, dmm%)
```

The input arguments to the `ibdev` subroutine are the board index number (`0`, for `GPIB0`), the primary GPIB address of the device (`1`), the secondary GPIB address of the device (`0`, for none), the timeout for the software to use when communicating with the device (`12`, for 3 sec), send END message with last data byte when writing to device (`1`, for enable), and EOS detection mode (`0`, for disable). When `ibdev` is called, the software automatically initializes the GPIB by sending an Interface Clear (IFC) message and places the device in its remote programming state.

Step 2. Clearing the Device

It is considered good practice to clear the device before you begin to configure the device for your application. Clearing the device resets its internal functions to a known state.

```
CALL ibclr(dmm%)
```

Step 3. Configuring the Device

After the `ibfind` and `ibclr` functions, the instrument is ready to receive commands. To configure the multimeter, device-specific commands are sent using the `ibwrt` function. The first argument of the `ibwrt` function is the unit descriptor for the meter returned in the variable `dmm%` by the `ibfind` function.

```
CALL ibwrt(dmm%, "F1R0S2T4")
```

The `ibwrt` function sends the bytes `F1R0S2T4` to the meter. The bytes sent to an instrument are different for most instruments. The command bytes for your instrument can be found in its user manual. The bytes in this example configure the voltage type (`F1`), voltage range (`R0`), update speed (`S2`), and the trigger mode (`T4`) of the meter.

Step 4. Triggering the Device

Previously, the device was set to wait for a trigger before sending a measurement reading. You must first send a trigger command to the device before reading the measurement value.

```
    CALL ibtrg(dmm%)
```

Step 5. Taking Measurements

Once the meter is configured, it can take a measurement and display it on its front panel. To read the measurement over the GPIB, the `ibrd` function is used. Again the first argument is the unit descriptor for the meter. The variable `rd$` holds the measurement value upon completion of the function. The meter sends 13 bytes of data across the GPIB. To allocate space for the measurement value, the `DIM` directive is used prior to the `ibrd` function.

```
    DIM RD AS STRING * 13
    CALL ibrd(dmm%, rd$)
```

The variable `rd$` holds the ASCII string representing the voltage measurement taken by the meter.

Step 6. Analyzing and Presenting the Acquired Data

Once the data has been acquired from the GPIB, all of the analysis and presentation functions of the programming language can be used to manipulate the data. Instead of reading just one voltage measurement from the meter, you could have read 10 values and calculated their average. The following code would replace the code in the last two lines of *Step 5. Taking Measurements*.

```
   DIM RD AS STRING * 13
   SUM = 0

   FOR X = 1 to 10
      CALL ibtrg(dmm%)
      CALL ibrd(dmm%, rd$)
      SUM = SUM + VAL(rd$)
   NEXT X

   PRINT "The average voltage is "; SUM/10
```

The Complete Application Program

The complete application program is as follows:

```
REM $INCLUDE: 'QBDECL.BAS'

CALL ibdev (0, 1, 0, 12, 1, 0, dmm%)
CALL ibclr(dmm%)

CALL ibwrt(dmm%, "F1R0S2T4")

DIM RD AS STRING * 13
SUM = 0

FOR X = 1 to 10
      CALL ibtrg(dmm%)
      CALL ibrd(dmm%, rd$)
      SUM = SUM + VAL(rd$)
NEXT X

PRINT "The average voltage is "; SUM/X

END
```

Compiling and Linking

After writing the application program in QuickBASIC, the program can be run within the QuickBASIC environment or as a stand-alone executable program. Both methods require the QuickBASIC language interface supplied on the distribution diskette. The following examples work for QuickBASIC 4.5.

Running a Program in the QuickBASIC Environment

Complete the following steps to run your program in the QuickBASIC environment.

1. Copy the QuickBASIC language interface (`QBIB.OBJ`) from the GPIB directory set up during installation into your QuickBASIC directory (`QB45` in this example, which stands for QuickBASIC version 4.5).

2. Create a QuickLibrary by entering the following command:

   ```
   C:\QB45> LINK /Q QBIB.OBJ,,,BQLB45.LIB
   ```

   This command creates a QuickLibrary named `QBIB.QLB`

3. Enter the QuickBASIC environment including this QuickLibrary by entering the following command:

   ```
   C:\QB45> QB /L QBIB.QLB
   ```

4. Load and run the application program.

Creating a Stand-alone Executable Program

Complete the following steps to create a stand-alone executable program.

1. Copy the QuickBASIC language interface (`QBIB.OBJ`) from the GPIB directory set up during installation into your QuickBASIC directory (`QB45` in this example).

2. Compile the application program (`AVGVOLTS.BAS`) by entering the following command:

   ```
   C:\QB45> BC /O AVGVOLTS.BAS;
   ```

3. Link the language interface by entering the following command:

   ```
   C:\QB45> LINK AVGVOLTS.OBJ+QBIB.OBJ;
   ```

   This step creates an executable file named AVGVOLTS.EXE.

4. Run the program by entering the following command:

   ```
   C:\QB45> AVGVOLTS
   ```

Equivalent Program Using Board Functions

The following QuickBASIC program is equivalent to the program just developed except that it uses board functions instead of device functions. The meter has a primary address of one (1) and thus has a talk address of hex 41 (ASCII A) and a listen address of hex 21 (ASCII !). The GPIB interface board has a primary address of zero (0) and thus has a talk address of hex 40 (ASCII @) and a listen address of hex 20 (ASCII <space>).

```
REM $INCLUDE: 'QBDECL.BAS'
CALL ibfind("gpib0", board%)
CALL ibsic(board%)
CALL ibsre(board%)

CALL ibcmd(board%, "!" + chr$(&H04))

CALL ibcmd(board%, "?_@!")
CALL ibwrt(board%, "F1R0S2T4")

DIM RD AS STRING * 13
SUM = 0

FOR X = 1 to 10
      CALL ibcmd(board%,"!" + chr$(&H08))
      CALL ibcmd(board%, "?_ A)
      CALL ibrd(board%, rd$)
      SUM = SUM + VAL(rd$)
NEXT X

PRINT "The average voltage is "; SUM/10

END
```

Equivalent BASIC Programs

The BASIC programs using device and board functions are equivalent to the QuickBASIC programs, with the exception of the declaration filename. `MBDECL.BAS` replaces `QBDECL.BAS`.

Equivalent C Program (Device Functions)

```
#include <stdio.h>
#include "decl.h"
main() {

      short dmm, x;
      double sum;
      char rd[13];

      dmm = ibdev (0, 1, 0, 12, 1, 0);
      ibclr (dmm);

      ibwrt (dmm, "F1R0S2T4", 8);

      for (sum=0, x=0; x <= 10; x++) {
       ibtrg (dmm);
       ibrd (dmm, rd, 13);
       sum = sum + atof(rd);
      }

      printf ("The average voltage is %f", sum/10);
}
```

Equivalent BASICA Program (Device Functions)

From the `BASICA` prompt, load `DECL.BAS` using the `BASICA MERGE` command and enter the program.

```
MERGE DECL.BAS
100 brdindx% = 0 : pad% = 1 : sad% = 0 : tmo% = 12
110 eot% = 1 : eos% = 0
120 CALL ibdev(brdindx%,pad%,sad%,tmo%,eot%,eos%,dmm%)
130 CALL ibclr(dmm%)
140 REM
145 wrt$ = "F1R0S2T4"
```

```
150 CALL ibwrt(dmm%, wrt$)
160 REM
170 SUM = 0
180 rd$ = space$(16)
190 FOR X = 1 to 10
200 CALL ibtrg(dmm%)
210 CALL ibrd(dmm%, rd$)
220 SUM = SUM + VAL(rd$)
230 NEXT X
240 REM
250 PRINT "The average voltage is "; SUM/10
260 REM
270 END
```

NI-488 Function Descriptions

The remainder of this chapter contains a detailed description of each NI-488 function with examples. The descriptions are listed alphabetically for easy reference.

IBBNA IBBNA

Purpose: Change access board of device.

Format:

BASICA

```
CALL ibbna (ud%, bname$)
```

QuickBASIC/BASIC

```
CALL ibbna (ud%, bname$)
```

or

```
ilbna (ud%, bname$)
```

C

```
short ibbna (short ud, char bname [])
```

`ud` specifies a device. `bname` specifies the new access board to be used in all device calls to that device. `ibbna` is needed only to alter the board assignment from its configuration setting.

The assigned board is used in all subsequent device functions used with that device until `ibbna` is called again, `ibonl` or `ibfind` is called, or the system is restarted.

When `ibbna` is called and an error does not occur, the previous access board number is stored in `ibcnt`.

IBBNA (continued) IBBNA

Device Function Example:

Associate the device `dvm` with the interface board `"GPIB0"`.

BASICA/QuickBASIC/BASIC

```
100  devname$ = "DEV10"
110  CALL ibfind (devname$, dvm%)
120  REM  Set GPIB0 as the access board for
130  REM  device dvm%.
140  bname$ = "GPIB0"
150  CALL ibbna (dvm%, bname$)
```

QuickBASIC/BASIC

```
dvm% = ilfind ("DEV10")
sta% = ilbna (dvm%, "GPIB0")
```

C

```
dvm = ibfind ("DEV10");
ibbna (dvm, "GPIB0");
```

.

IBCAC IBCAC

Purpose: Become Active Controller.

Format:

BASICA

```
CALL ibcac (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibcac (ud%, v%)
          or
ilcac (ud%, v%)
```

C

short `ibcac` (short `ud`, short `v`)

`ud` specifies an interface board. If `v` is non-zero, the GPIB board takes control synchronously with respect to data transfer operations; otherwise, the GPIB board takes control immediately (asynchronously).

To take control synchronously, the GPIB board asserts the ATN signal without corrupting data being transferred. If a data handshake is in progress, the take control action is postponed until the handshake is complete. If a handshake is not in progress, the take control action is done immediately. Synchronous take control is not guaranteed if an `ibrd` or `ibwrt` operation completed with a timeout or error.

Asynchronous take control should be used in situations where it appears to be impossible to gain control synchronously (for example, after a timeout error).

It is generally not necessary to use the `ibcac` function in most applications. Functions, such as `ibcmd` and `ibrpp`, that require the GPIB board to take control, do so automatically.

The ECIC error results if the GPIB board is not CIC.

IBCAC (continued) IBCAC

Board Function Example:

1. Take control immediately without regard to transfers in progress.

   BASICA/QuickBASIC/BASIC

```
100  v% = 0
110  CALL ibcac (brd0%, v%)
120  REM  ibsta% should show board is now CAC
130  REM  (CIC ATN).
```

   QuickBASIC/BASIC

```
sta% = ilcac (brd0%, 0)
```

   C

```
ibcac (brd0, 0);
```

2. Take control synchronously and assert ATN following a read operation.

   BASICA/QuickBASIC/BASIC

```
100  board$ = "GPIB0"
110  CALL ibfind (board$, brd0%)
120  CALL ibrd (brd0%, rd$)
130  v% = 1
140  CALL ibcac (brd0%, v%)
```

   QuickBASIC/BASIC

```
brd0% = ilfind ("GPIB0")
sta% = ilrd (brd0%, rd$, 512)
sta% = ilcac (brd0%, 1)
```

IBCAC (continued) IBCAC

C

```
short brd0;
brd0 = ibfind ("gpib0");
ibrd (brd0,rd,512);
ibcac (brd0,1);
```

IBCLR IBCLR

Purpose: Clear specified device.

Format:

BASICA

```
CALL ibclr (ud%)
```

QuickBASIC/BASIC

```
CALL ibclr (ud%)
      or
ilclr (ud%)
```

C

short `ibclr` (short `ud`)

`ud` specifies a device.

The `ibclr` function clears the internal or device functions of a specified device.

`ibclr` calls the board function `ibcmd` to send the following commands using the designated access board:

- Talk address of access board
- Unlisten (UNL)
- Listen address of the device
- Secondary address of the device, if applicable
- Selected Device Clear (SDC)

Other command bytes may be sent as necessary.

Refer to *IBCMD* for additional information. Refer also to the discussion of device functions in Chapter 3, *Understanding the NI-488.2 Software*.

IBCLR (continued) IBCLR

Device Function Example:

Clear the device `vmtr`.

BASICA/QuickBASIC/BASIC

```
100  dev$ = "DEV3" 'open the voltmeter
110  CALL ibfind (dev$, vmtr%)
120  REM Clear the voltmeter
130  CALL ibclr (vmtr%)
```

QuickBASIC/BASIC

```
' Open instrument
vmtr% = ilfind ("dev3")
' Clear it
sta% = ilclr (vmtr%)
```

C

```
short vmtr;

vmtr = ibfind ("dev3"); /* open instrument */
ibclr (vmtr); /* clear it */
```

IBCMD IBCMD

Purpose: Send GPIB command messages.

Format:

BASICA

```
CALL ibcmd (ud%, cmd$)
```

QuickBASIC/BASIC

```
CALL ibcmd (ud%, cmd$)
     or
ilcmd (ud%, cmd$, cnt&)
```

C

short `ibcmd` (short `ud`, char `cmd` [], unsigned long `cnt`)

`ud` specifies an interface board. `cmd` contains the commands to be sent over the GPIB.

The `ibcmd` function is used to transmit interface messages (commands) over the GPIB. These commands are listed in Appendix A, *Multiline Interface Messages*. The `ibcmd` function is also used to pass GPIB control to another device. This function is *not* used to transmit programming instructions to devices. These instructions are transmitted with the `ibrd` and `ibwrt` functions.

The `ibcmd` operation terminates on any of the following events:

- All commands are successfully transferred.
- An error is detected.
- The time limit is exceeded.
- A Take Control (TCT) command is sent.
- An Interface Clear (IFC) message is received from the System Controller.

IBCMD (continued) IBCMD

The transfer count may be less than the requested count on any of the previous terminating events but the first.

In C, the requested transfer count, `cnt`, can be any value that fits into a long integer, although command transfers are typically very small. To more easily accommodate a small count from an application program, the C language interface allows an integer value to be passed to the function in place of a long value without any problem.

An ECIC error results if the GPIB board is not CIC. If it is not Active Controller, the GPIB board takes control and asserts ATN prior to sending the command bytes. The GPIB board remains Active Controller afterward.

In the examples that follow, GPIB commands and addresses are coded as printable ASCII characters. If values correspond to printable ASCII characters, it is simplest to use the ASCII characters to specify the values. Refer to Appendix A, *Multiline Interface Messages*, for the ASCII characters corresponding to a numeric value.

Board Function Examples:

1. Unaddress all Listeners (UNL or ASCII ?) and address a Talker at hex 46 (ASCII F) and a Listener at hex 31 (ASCII 1).

   BASICA/QuickBASIC/BASIC

   ```
   100  cmd$ = "?F1"        ' UNL TAD LAD
   110  CALL ibcmd (brd0%, cmd$)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilcmd (brd0%, "?F1", 3)
   ```

   C

   ```
   ibcmd (brd0, "?F1", 3); /* UNL TAD LAD         */
   ```

IBCMD (continued) IBCMD

2. Same as Example 1, except the Listener has a secondary address of hex 6E (ASCII n).

   BASICA/QuickBASIC/BASIC

   ```
   100  cmd$ = "?F1n"      ' UNL TAD LAD SAD
   110  CALL ibcmd (brd0%, cmd$)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilcmd (brd0%,"?F1n",4)
   ```

   C

   ```
   ibcmd (brd0,"?F1n",4);  /*  UNL TAD LAD SAD   */
   ```

3. Clear all GPIB devices with the Device Clear (DCL or hex 14) command.

   BASICA/QuickBASIC/BASIC

   ```
   100  cmd$ = chr$(&H14)   ' DCL
   110  CALL ibcmd (brd0%, cmd$)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilcmd (brd0%, chr$(&H14), 1)
   ```

   C

   ```
   ibcmd (brd0, "\x14", 1);    /*         DCL    */
   ```

4. Clear two devices with listen addresses of hex 21 (ASCII !) and hex 28 (ASCII ([left parenthesis]) with the Selected Device Clear (SDC or hex 04) command).

   BASICA/QuickBASIC/BASIC

   ```
   100  cmd$ = "?!(" + chr$(&H04) ' UNL LAD LAD SDC
   110  CALL ibcmd (brd0%, cmd$)
   ```

IBCMD (continued) IBCMD

QuickBASIC/BASIC

```
sta% = ilcmd (brd0%, "?!(" + chr$(&H04), 4)
```

C

```
ibcmd (brd0,"?!(\x04", 4); /* UNL LAD LAD SDC */
```

5. Trigger any devices previously addressed to listen with the Group Execute Trigger (GET or hex 08) command.

BASICA/QuickBASIC/BASIC

```
100  cmd$ = chr$(&H08)   ' GET
110  CALL ibcmd (brd0%, cmd$)
```

QuickBASIC/BASIC

```
sta% = ilcmd (brd0%, chr$(&H08), 1)
```

C

```
ibcmd (brd0, "\x08", 1);  /*  GET              */
```

IBCMD (continued) IBCMD

6. Serial poll a device at talk address hex 52 (ASCII `R`) using the Serial Poll Enable (SPE or hex 18) and Serial Poll Disable (SPD or hex 19) commands (the GPIB listen address is hex 20 or ASCII <space>).

   BASICA/QuickBASIC/BASIC

```
100  cmd$ ="R " + chr$(&H18)  'TAD MLA SPE
110  CALL ibcmd (brd0%, cmd$)
120  rd$ = space$(1)
130  CALL ibrd (brd0%, rd$)
140  REM  After checking the status byte in
150  REM  rd$, disable this device and
160  REM  unaddress it with the Untalk
170  REM  (UNT or ASCII _) command before
180  REM  polling the next one.
190  cmd$ = chr$(&H19) + "_"     ' SPD UNT
200  CALL ibcmd (brd0%, cmd$)
```

   QuickBASIC/BASIC

```
sta% = ilcmd (brd0%, "R_" + chr$(&H18), 3)
rd$ = space$(1)
sta% = ilrd (brd0%, rd$, 1)
sta% = ilcmd (brd0%, chr$(&H19) + "_", 2)
```

   C

```
ibcmd (brd0, "R \x18", 4);  /* TAD MLA SPE    */
ibrd (brd0, rd, 1);    /*   read one byte    */
ibcmd (brd0, "\x19_", 2);     /*   SPD UNT    */
```

IBCMDA IBCMDA

Purpose: Send commands asynchronously from string.

Format:

BASICA

```
CALL ibcmda (ud%, cmd$)
```

QuickBASIC/BASIC

```
CALL ibcmda (ud%, cmd$)
```

or

```
ilcmda (ud%, cmd$, cnt&)
```

C

short `ibcmda` (short `ud`, char `cmd` [], unsigned long `cnt`)

`ud` specifies an interface board. `cmd` contains the commands to be sent over the GPIB.

The `ibcmda` function is used to transmit interface messages (commands) over the GPIB. These commands are listed in Appendix A, *Multiline Interface Messages*. The `ibcmda` function can also be used to pass GPIB control to another device. This function is *not* used to transmit programming instructions to devices. These instructions and other device-dependent information are transmitted with the `ibrd` and `ibwrt` functions.

`ibcmda` is used in place of `ibcmd` if the application program must perform other functions while processing the GPIB command. `ibcmda` returns immediately after starting the I/O operation.

The three asynchronous I/O calls (`ibcmda`, `ibrda`, and `ibwrta`) are designed to allow an application to perform other functions (non-GPIB functions) while processing the I/O. Once the asynchronous I/O call has been initiated, further GPIB calls involving the device or access board are not allowed until the I/O has completed and the GPIB software and the application have been resynchronized.

IBCMDA (continued) IBCMDA

Resynchronization can be accomplished by using one of the following three functions:

Note: Resynchronization is only successful if the `ibsta` returned contains CMPL.

- `ibwait` - The driver and application are synchronized. The wait mask is arbitrary.
- `ibstop` - The asynchronous I/O is canceled, and the driver and application are synchronized.
- `ibonl` - The asynchronous I/O is canceled, the interface has been reset, and the driver and application are synchronized.

Any other GPIB call involving the device or access board returns the EOIP error.

An ECIC error results if the GPIB board is not CIC. If it is not Active Controller, the GPIB board takes control and asserts ATN prior to sending the command bytes. It remains Active Controller afterward. The ENOL error is returned if there are no other devices on the IEEE-488 bus.

IBCMDA (continued) IBCMDA

Board Function Example:

Address several devices for a broadcast message to follow while testing for a high-priority event to occur.

BASICA/QuickBASIC/BASIC

```
100  REM  The interface board BRD0% at talk
110  REM  address &H40 (ASCII @), addresses
120  REM  nine Listeners at addresses &H31-
130  REM  &H39 (ASCII 1-9) to receive the
140  REM  broadcast message.
150  board$ = "GPIB0"
160  CALL ibfind (board$, brd0%)
170  cmd$ = "?@123456789"      ' UNL MTA
180                            ' LAD1...LAD9
190  CALL ibcmda (brd0%, cmd$)
200  GOSUB 1000                ' Unspecified routine
210                            ' for a high priority
220                            ' event.
230  mask% = 0                 ' Set mask to return
240                            ' immediately.
250  CALL ibwait (brd0%, mask%)
260  IF ibsta% AND TIMO THEN GOTO 900 'jump on error
270  IF (ibsta% AND &H100) <> &H100 GOTO 200
280  PRINT "Asynchronous commands sent!"
290  mask% = &H4100
300  CALL ibwait (brd0%,mask%)
310  PRINT "Asynchronous transfer properly
terminated"
320  …
900  PRINT "Error" : STOP
910  END
```

IBCMDA (continued) IBCMDA

QuickBASIC/BASIC

```
brd0% = ilfind ("GPIB0")
ilcmda (brd0%,"?@123456789",11)
sta% = 0
While (sta% AND &H100) <> &H100
       CALL eventtest
       sta% = ilwait (brd0%, 0)
       IF sta% < 0 THEN GOSUB ERROR
WEND
PRINT "Asynchronous commands sent!"
sta% = ilwait (brd0%,&H4100)
PRINT "Asynchronous transfer properly terminated"
```

C

```
brd0 = ibfind ("GPIB0");
ibcmda (brd0, "?@123456789", 11);
sta = 0;
while ((sta & 0x100) != 0x100) {
   event_test();
   ibwait (brd0, 0);
   sta = ibsta;
   if (sta < 0) error ();
}
printf ("Asynchronous commands sent!");
ibwait (brd0,0x4100)
printf ("Asynchronous transfer properly terminated");
```

IBCONFIG IBCONFIG

Purpose: Change the software configuration parameters.

Format:

BASICA

```
CALL ibconfig (ud%, option%, value%)
```

QuickBASIC/BASIC

```
CALL ibconfig (ud%, option%, value%)
```

or

```
ilconfig (ud%, option%, value%)
```

C

short `ibconfig` (short `ud,` unsigned short `option,` unsigned short `cnt)`

`ud` specifies a GPIB interface board or a device. `option` is used to select the configurable item in the software. The configurable item is set to the contents of `value`. The previous contents of the configurable item are returned in `iberr`. If `ud` is a GPIB interface board descriptor, `option` takes on the values shown in Table 5-4. If `ud` is a device descriptor, `option` has the values shown in Table 5-5.

Table 5-4. Board Configuration Options

Option	Description
`1`	Primary Address. `value` is the new primary address of the GPIB interface board (0–30). See *IBPAD* and Appendix A.
`2`	Secondary Address. `value` is the new secondary address of the board (0, 96–126). See *IBSAD* and Appendix A.

(continues)

IBCONFIG (continued) IBCONFIG

Table 5-4. Board Configuration Options (continued)

Option	Description
`3`	Timeout Value. `value` is the new timeout value of the board (0–15). See *IBTMO*.
`4`	Enable/disable END message on write operations. `value` is the new EOT mode (0 = no END, non-zero = send END with last byte). See *IBEOT*.
`5`	Parallel Poll Configure. `value` is the parallel poll configure byte (0, 96–126). See *IBPPC*.
`7`	Enable/disable Automatic Serial Polling. If `value` is zero (0), Autopolling is disabled. If `value` is non-zero, Autopolling is enabled.
`8`	Use/do not use the NI-488 CIC protocol. If `value` is zero (0), do not use the CIC protocol. If `value` is non-zero, use the CIC protocol. See the *Device Functions* section earlier in this chapter.
`9`	Enable/disable hardware interrupts. If `value` is zero (0), disable GPIB interface board interrupts. If `value` is non-zero, enable GPIB interface board interrupts. See description of the `IBCONF` utility program in Chapter 2.
`10`	Request or release System Control. If `value` is zero (0), functions requiring System Controller capability are not allowed. If `value` is non-zero, functions requiring System Controller capability are allowed. See *IBRSC*.
`11`	Assert/unassert REN. If `value` is non-zero, the IEEE-488 Remote Enable (REN) signal is asserted. If `value` is zero (0), REN is unasserted. See *IBSRE*.

(continues)

IBCONFIG (continued) IBCONFIG

Table 5-4. Board Configuration Options (continued)

Option	Description
`12`	Terminate read when End-of-String (EOS) character is detected. If `value` is non-zero, read functions are terminated when the EOS character is detected in the data stream. If `value` is zero, EOS detection is disabled. See *IBEOS*.
`13`	Assert EOI when sending EOS character. If `value` is zero (0), do not send EOI with EOS. If `value` is non-zero, send EOI with EOS. See *IBEOS*.
`14`	Use 7- / 8-bit EOS comparison. If `value` is zero, use low-order 7 bits of EOS character for comparison. If `value` is non-zero, use 8 bits. See *IBEOS*.
`15`	End-Of-String (EOS) character. `value` is the new EOS character of the board (8 bits). See *IBEOS*.
`16`	Parallel Poll remote/local configuration. If `value` is zero, the GPIB interface board uses IEEE-488 Parallel Poll (PP) interface function subset PP1 (remote configuration by external Controller). If `value` is non-zero, the board uses PP subset PP2 (local configuration from your application program: `value` is used as the local poll enable [lpe] message). See *IBPPC*.
`17`	IEEE-488 bus handshake timing. If `value` is one (1), normal timing is used for the IEEE-488 Source Handshake T1 delay (≥ 2 µsec). If `value` is two (2), high-speed timing is used for T1 (≥ 500 nsec). If `value` is three (3), very high-speed timing is used (≥ 350 nsec).

(continues)

IBCONFIG (continued) IBCONFIG

Table 5-4. Board Configuration Options (continued)

Option	Description
18	Enable/disable direct memory access (DMA) transfers. If `value` is zero (0), disable GPIB interface board DMA transfers. If `value` is non-zero, enable GPIB interface board DMA transfers. See the description of the `IBCONF` utility program in Chapter 2.
19	Byte swapping on `ibrd`. If `value` is one (1), pairs of bytes read off the bus are swapped before storing them in the `ibrd` buffer. The transfer count must be even or ECAP is returned. In this case, the last two bytes of the buffer are invalid. If ECAP is returned and your buffer begins on an odd address, start the buffer on an even address. If `value` is zero (0), byte swapping on `ibrd` is disabled.
20	Byte swapping on `ibwrt`. If `value` is one (1), pairs of bytes are swapped before they are written from the user's buffer to the bus. The transfer count must be even or ECAP is returned. In some cases, the address of the buffer must be even. If ECAP is returned and your buffer begins on an odd address, start the buffer on an even address. If `value` is zero (0), byte swapping on `ibwrt` is disabled.
21	Enable/disable the software's event queue. If `value` is one (1), the event queue is enabled, the EVENT bit in `IBSTA` becomes active, and the DTAS and DCAS bits become inactive. When the queue is enabled, any device clear or trigger message is placed in the queue, the EVENT bit is set, and any I/O operation in progress is aborted. The `ibevent` function can be used to retrieve the events. If `value` is zero (0), the event queue is disabled. This is the default. This option can be used only if hardware interrupts are enabled for the board.

(continues)

IBCONFIG (continued) IBCONFIG

Table 5-4. Board Configuration Options (continued)

Option	Description
`22`	Enable/disable the SPOLL bit of `IBSTA`. If `value` is one (1), the SPOLL bit becomes active. It is then set whenever the interface board is serial polled by the Controller and the board is requesting service. If `value` is zero (0), the default, the SPOLL bit is not active. This option can be used only if hardware interrupts are enabled for the board.
`23`	Enable/disable the sending of LLO when putting a device online. If `value` is zero (0), the default, Local Lockout is not sent to the remote device when `ibfind` is used. If you want the device to automatically go into Local Lockout state, set `value` to one (1) before calling `ibfind` for any devices.
`25`	Set the length of time that the driver waits for a parallel poll response. The default length of 2 µsec is selected by setting `value` to zero (0). To select another poll length, use the same timing values as the `ibtmo` function. The legal range is 0 to 15.
`26`	Enable/disable the setting of the END bit in `IBSTA` on receiving the End-of-String (EOS) character. If `value` is one (1), the default, the END bit of `IBSTA` is set whenever EOI, EOS, or EOI plus EOS is received. If `value` is zero (0), the END bit is set only when EOI or EOI plus EOS is received. If the EOS character is received without EOI, the END bit is not set. This option can be useful in applications that need to distinguish between the three I/O read termination modes (EOI by itself, EOS by itself, and EOI plus the EOS character).

IBCONFIG (continued) IBCONFIG

Table 5-5. Device Configuration Options

Option	Description
`1`	Primary Address. `value` is the new primary address of the device (0–30). See *IBPAD* and Appendix A.
`2`	Secondary Address. `value` is the new secondary address of the device (0, 96–126). See *IBSAD* and Appendix A.
`3`	Timeout Value. `value` is the new timeout value of the device (0–15). See *IBTMO*.
`4`	Enable/disable END message on write operations. `value` is the new EOT mode (0 = no END, non-zero = send END with last byte). See *IBEOT*.
`6`	Repeat Addressing. If `value` is zero (0), disable repeat addressing. If `value` is non-zero (1), enable repeat addressing. See the description of the `IBCONF` utility in Chapter 2.
`12`	Terminate read when End-of-String (EOS) character is detected from this device. If `value` is non-zero, read functions are terminated when the EOS character is detected in the data stream received from the device. If `value` is zero, EOS detection is disabled. See *IBEOS*.
`13`	Assert EOI when sending EOS character to this device. If `value` is zero (0), do not send EOI with EOS. If `value` is non-zero, send EOI with EOS. See *IBEOS*.
`14`	Use 7- / 8-bit EOS comparison. If `value` is zero, use low-order 7 bits of EOS character for comparison. If `value` is non-zero, use 8 bits. See *IBEOS*.

(continues)

IBCONFIG (continued) IBCONFIG

Table 5-5. Device Configuration Options (continued)

Option	Description
`15`	End-of-String (EOS) character. `value` is the new EOS character (8 bits) to use with this device. See *IBEOS*.
`19`	Byte swapping on `ibrd`. If `value` is one (1), pairs of bytes read off the bus are swapped before storing them in the `ibrd` buffer. The transfer count must be even or ECAP is returned. In this case, the last two bytes of the buffer are invalid. If ECAP is returned and your buffer begins on an odd address, start the buffer on an even address. If `value` is zero (0), byte swapping on `ibrd` is disabled.
`20`	Byte swapping on `ibwrt`. If `value` is one (1), pairs of bytes are swapped before they are written from the user's buffer to the bus. The transfer count must be even or ECAP is returned. In some cases, the address of the buffer must be even. If ECAP is returned and your buffer begins on an odd address, start the buffer on an even address. If `value` is zero (0), byte swapping on `ibwrt` is disabled.
`24`	Set the serial poll timeout length. The IEEE-488.1 specification does not specify the length of time that a Controller should wait for a response byte. The software's default time of 1 second (`value` is 11) is sufficient for most devices. If you need a longer delay, set `value` to one of the `ibtmo` timeout values. The legal range is 0 to 15.
`27`	Enable/disable the automatic sending of the bus commands untalk (UNT) and unlisten (UNL) after each read or write operation. If value is zero (0), the default, then UNT and UNL are not sent after each I/O operation. This mode is IEEE-488.2 compliant. If the value is one (1), then the IEEE-488.1 bus commands UNT and UNL are sent onto the bus after every I/O operation. This mode is not IEEE-488.2 compliant, but may be needed when communicating with non-488.2 devices.

IBCONFIG (continued) IBCONFIG

Device Function Examples

Set up various configurable parameters in preparation for a device read.

BASICA/QuickBASIC/BASIC

```
100  dev$ = "dev1"
110  CALL ibfind (dev$, dev1%)
120  REM Enable repeat addressing
130  option% = 6 : value% = 1
140  CALL ibconfig (dev1%, option%, value%)
150  REM Set linefeed as the EOS character
160  option% = 15 : value% = 10
170  CALL ibconfig (dev1%, option%, value%)
180  REM Use 7-bit comparison for EOS character
190  option% = 14 : value% = 0
200  CALL ibconfig (dev1%, option%, value%)
210  REM Terminate reads on EOS
220  option% = 12 : value% = 1
230  CALL ibconfig (dev1%, option%, value%)
```

QuickBASIC/BASIC

```
' open device
CALL ibfind ("dev1", dev1%)
' Enable repeat addressing
CALL ibconfig (dev1%, 6, 1)
' Set linefeed as the EOS character
CALL ibconfig (dev1%, 15, 10)
' Use 7-bit comparison for EOS character
CALL ibconfig (dev1%, 14, 0)
' Terminate reads on EOS
CALL ibconfig (dev1%, 12, 1)
```

IBCONFIG (continued) IBCONFIG

C

```
/* open device */
dev1 = ibfind ("dev1");
/* Enable repeat addressing */
ibconfig (dev1, 6, 1)
/* Set linefeed as the EOS character */
ibconfig (dev1, 15, 10)
/* Use 7-bit comparison for EOS character */
ibconfig (dev1, 14, 0)
/* Terminate reads on EOS */
ibconfig (dev1, 12, 1)
```

Board Function Examples:

1. Set up various configurable parameters in preparation for a board read.

   BASICA/QuickBASIC/BASIC

```
100  gpib$ = "gpib0"
110  CALL ibfind (gpib$, gpib0%)
120  REM Enable DMA transfers
130  option% = 18 : value% = 1
140  CALL ibconfig (gpib0%, option%, value%)
150  REM Turn off Autopolling
160  option% = 7 : value% = 0
170  CALL ibconfig (gpib0%, option%, value%)
180  REM Turn on interrupts
190  option% = 9 : value% = 1
200  CALL ibconfig (gpib0%, option%, value%)
```

IBCONFIG (continued) IBCONFIG

QuickBASIC/BASIC

```
' open GPIB interface board
CALL ibfind ("gpib0", gpib0%)
' Enable DMA transfers
CALL ibconfig (gpib0%, 18, 1)
' Turn off Autopolling
CALL ibconfig (gpib0%, 7, 0)
' Turn on interrupts
CALL ibconfig (gpib0%, 9, 1)
```

C

```
/* open GPIB interface board */
gpib0 = ibfind ("gpib0");
/* Enable DMA transfers */
ibconfig (gpib0, IbcDMA, 1)
/* Turn off Autopolling */
ibconfig (gpib0, IbcAUTOPOLL, 0)
/* Turn on interrupts */
ibconfig (gpib0, IbcIRQ, 1)
```

IBCONFIG (continued) IBCONFIG

2. Enable automatic byte swapping of binary integer data.

BASICA/QuickBASIC/BASIC

```
50   DIM array% (500)
100  REM Read in unswapped header data
101  REM (10 bytes).
102  header$ = space$(10)
103  CALL ibrd (ud%, header$)
104  REM Arrange for byte swapping.
105  option% = 19 : value% = 1
106  CALL ibconfig (ud%, option%, value%)
107  REM Read 1,000 bytes with automatic
108  REM swapping.
109  count% = 1000
110  CALL ibrdi (ud%, array% (0), count%)
111  REM Disable swapping for subsequent reads.
112  value% = 0
113  CALL ibconfig (ud%, option%, value%)
```

QuickBASIC/BASIC

```
DIM array% (500)
' Read in unswapped header data (10 bytes).
header$ = space$(10)
sta% = ilrd (ud%, header$, 10)
' Arrange for byte swapping.
sta% = ilconfig (ud%, 19, 1)
' Read 1,000 bytes with automatic swapping.
sta% = ilrdi (ud%, array% (), 1000)
' Disable swapping for subsequent reads.
sta% = ilconfig (ud%, 19, 0)
```

IBCONFIG (continued) IBCONFIG

C

```
short array [500]
char header [10]

/* Read in unswapped header data. */
ibrd (ud, header, 10)
/* Arrange for byte swapping. */
ibconfig (ud, 19, 1)
/* Read 1,000 bytes with automatic swapping. */
ibrdi (ud, array, 1000)
/* Disable swapping for subsequent reads. */
ibconfig (ud, 19, 0)
```

IBDEV IBDEV

Purpose: Open and initialize an unused device when the device name is unknown.

Format:

BASICA

```
CALL ibdev (board.index%,pad%,sad%,tmo%,eot%,eos%,ud%)
```

QuickBASIC/BASIC

```
CALL ibdev (board.index%,pad%,sad%,tmo%,eot%,eos%,ud%)
```

or

```
 ud% = ildev (board.index%,pad%,sad%, tmo%,eot%,eos%)
```

C

```
ud = short ibdev (short boardindex, short pad, short sad,
                  short tmo, short eot, short eos)
```

`boardindex` is an index from 0 to [(number of boards) - 1] of the access board with which the device descriptor must be associated. The arguments `pad`, `sad`, `tmo`, `eot`, and `eos` dynamically set the software configuration for the NI-488 I/O functions. These arguments configure the primary address, secondary address, I/O timeout, asserting EOI on last byte of data sourced, and the End-Of-String mode and byte, respectively. Refer to `IBPAD`, `IBSAD`, `IBTMO`, `IBEOT`, and `IBEOS` for more information on each argument. The device descriptor is returned in the variable `ud`.

The `ibdev` command selects an unopened device, opens it, and initializes it. You can use this function in place of `ibfind`.

`ibdev` returns a device descriptor of the first unopened user-configurable device that it finds. For this reason, it is very important to use `ibdev` *only after* all of your `ibfind` calls have been made. This is the only way to ensure that `ibdev` does not use a device that you plan to use via an `ibfind` call. The `ibdev` function performs the equivalent of the `ibonl` function to open the device.

IBDEV (continued) IBDEV

Note: The device descriptor of the NI-488.2 driver can remain open across invocations of an application, so be sure to return the device descriptor to the pool of available devices by calling `ibonl` with `v`=0 when you are finished using the device. If you do not, that device will not be available for the next `ibdev` call.

If the `ibdev` call fails, a negative number is returned in place of the device descriptor. There are two distinct errors that can occur with the `ibdev` call:

- If no device is available or the specified board index refers to a non-existent board, `ibdev` returns the EDVR or ENEB error.
- If one of the last five parameters is an illegal value, `ibdev` returns with a good board descriptor and the EARG error.

Device Function Example:

1. `ibdev` opens an available device and assigns it to access `GPIB0` (`board` = 0) with a primary address of 6 (`pad` = 6), a secondary address of 0x67 (`sad` = 0x67), a timeout of 10 msec (`tmo` = 7), the END message enabled (`eot` = 1) and the EOS mode disabled (`eos` = 0).

   BASICA/QuickBASIC/BASIC

```
100  REM  Get a device descriptor associated
105  REM  with board 0 (GPIB0).
110             board% = 0
115             pad% = 6
120             sad% = &H67
125             tmo% = 7
130             eot% = 1
135             eos% = 0
140  CALL ibdev(board%,pad%,sad%,tmo%,eot%,eos%,ud%)
150  REM  If ud% is less than 0, you have an
155  REM  error and you cannot continue with
156  REM  your program.
```

IBDEV (continued) IBDEV

QuickBASIC/BASIC

```
      ud% = ildev(0,6,&H67,7,1,0)
'  If ud% is less than 0, you have an error
'  and you cannot continue with your program.
```

C

```
if ((ud = ibdev(0,6,0x67,7,1,0)) < 0) {
     /* Handle GPIB error here */
     if (iberr == EDVR) {
          /* bad boardindex or no devices
           * available.
           */
     }
}
else if (iberr == EARG) {
     /* the call succeeded, but at least one
      * of pad,sad,tmo,eos,eot is incorrect.
      */
}
```

IBDMA IBDMA

Purpose: Enable or disable DMA.

Format:

BASICA

```
CALL ibdma (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibdma (ud%, v%)
        or
ildma (ud%, v%)
```

C

short `ibdma` (short `ud`, short `v`)

`ud` specifies an interface board. If `v` is non-zero, DMA transfers between the GPIB board and memory are used for read and write operations. If `v` is zero, programmed I/O is used.

If you enabled DMA at configuration time, this function can be used to switch between programmed I/O and the selected DMA channel. If you disabled DMA at configuration time or your computer does not have DMA capability, calling this function with `v` equal to a non-zero value results in an ECAP error.

The assignment made by this function remains in effect until `ibdma` is called again, the `ibonl` or `ibfind` function is called, or the system is restarted.

When `ibdma` is called and an error does not occur, the previous value of `v` is stored in `iberr`.

IBDMA **(continued)** **IBDMA**

Board Function Examples:

1. Enable DMA transfers using the previously configured channel.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = 1    ' Any non-zero value will do.
   110  CALL ibdma (brd0%, v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ildma (brd0%, 1)
   ```

   C

   ```
   ibdma(brd0, 1); /* Any non-zero value will do. */
   ```

2. Disable DMA and use programmed I/O exclusively.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = 0
   110  CALL ibdma (brd0%, v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ildma (brd0%, 0)
   ```

   C

   ```
   ibdma (brd0, 0);
   ```

IBEOS IBEOS

Purpose: Change or disable End-of-String termination mode.

Format:

BASICA

```
CALL ibeos (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibeos (ud%, v%)
   or
ileos (ud%, v%)
```

C

short `ibeos` (short `ud`, short `v`)

`ud` specifies a device or an interface board. `v` specifies the EOS character and the data transfer termination method according to Table 5-6. `ibeos` is needed only to alter the value from its configuration setting.

The assignment made by this function remains in effect until `ibeos` is called again, the `ibonl` or `ibfind` function is called, or the system is restarted.

When `ibeos` is called and an error does not occur, the previous value of `v` is stored in `iberr`.

IBEOS (continued) IBEOS

Table 5-6. Data Transfer Termination Method

	Value of v	
Method	**High Byte**	**Low Byte**
A. Terminate read when EOS is detected.	00000100	EOS
B. Set EOI with EOS on write function.	00001000	EOS
C. Compare all 8 bits of EOS byte rather than low 7 bits (all read and write functions).	00010000	EOS

Methods A and C determine how read operations terminate. If Method A alone is chosen, reads terminate when the low seven bits of the byte that is read match the low seven bits of the EOS character. If Methods A and C are chosen, a full 8-bit comparison is used.

Methods B and C together determine when write operations send the END message. If Method B alone is chosen, the END message is sent automatically with the EOS byte when the low seven bits of that byte match the low seven bits of the EOS character. If Methods B and C are chosen, a full 8-bit comparison is used.

Note: Defining an EOS byte for a device or board does not cause the driver to automatically send that byte when performing writes. Your application program must include the EOS byte in the data string it defines.

Device IBEOS Function

If `ud` specifies a device, the options coded in `v` are used for all device reads and writes in which that device is specified.

Board IBEOS Function

If `ud` specifies a board, the options coded in `v` become associated with all board reads and writes.

IBEOS (continued) IBEOS

Refer also to *IBEOT*.

Device Function Example:

Send END when the linefeed character is written to the device `dvm`.

BASICA/QuickBASIC/BASIC

```
10   EOSV% = &H0A       ' EOS info for ibeos.
:
:
100  v% = EOSV% + &H0800
110  CALL ibeos (dvm%, v%)
120  wrt$ = "123" + chr$(&H0A)
150                     ' EOS character is the
160                     ' last byte of string.
170  CALL ibwrt (dvm%, wrt$)
```

QuickBASIC/BASIC

```
EOSV% = &H0A
.
.
.
sta% = ileos (dvm%, EOSV% + &H0800)
ilwrt (dvm%, "123" + chr$(&H0A), 4)
```

C

```
v = XEOS | '\n'; /* EOS information for ibeos.*/
ibeos (dvm, v);
ibwrt (dvm, "123\n", 4);
```

IBEOS (continued) IBEOS

Board Function Examples:

1. Program the interface board `brd0` to terminate a read on detection of the linefeed character (hex 0A) that is received within 200 bytes.

   BASICA/QuickBASIC/BASIC

```
10   EOSV% = &H0A
:
:
100  v% = EOSV% + &H0400
110  CALL ibeos (brd0%, v%)
120  REM  Assume board has been addressed; do
130  REM  board read.
140  rd$ = space$(200)
150  CALL ibrd (brd0%, rd$)
160  REM  The END bit in ibsta% is set if the
170  REM  read terminated on the EOS.
```

   QuickBASIC/BASIC

```
EOSV% = &H0A
.
.
.
sta% = ileos (brd0%, EOSV% + &H0400)
rd$ = space$(200)
sta% = ilrd (brd0%, rd$, 200)
```

   C

```
char rd[200];

v = REOS | '\n';
ibeos (brd0, v);
ibrd (brd0, rd, 200);
```

IBEOS (continued) IBEOS

2. To program the interface board `brd0` to terminate read operations on the 8-bit value hex 82 rather than the 7-bit character hex 0A, change lines 10 and 100 in Example 1.

   BASICA/QuickBASIC/BASIC

```
10   EOSV% = &H82
:
:
100  v% = EOSV% + &H1400
:
:
```

   QuickBASIC/BASIC

```
EOSV% = &H82
.
.
.
sta% = ileos(brd0%, EOSV% + &H1400)
```

   C

```
v = BIN | REOS | 0x82;
ibeos (brd0, v);
```

3. To disable read termination on receiving the EOS character, change line 100 in Example 1.

   BASICA/QuickBASIC/BASIC

```
:
:
100  v% = EOSV%
```

IBEOS (continued) IBEOS

QuickBASIC/BASIC

```
.
.
.
sta% = ileos (brd0%, EOSV%)
```

C

```
v= '\n';
.
.
.
ibeos (brd0, v);
```

4. Send END when the linefeed character is written.

BASICA/QuickBASIC/BASIC

```
10   EOSV% = &H0A    ' EOS info for IBEOS.
:
:
100  v% = EOSV% + &H0800
110  CALL ibeos (brd0%, v%)
120  REM  Assume the board has been
130  REM  addressed; do board write.
140  wrt$ = "123" + chr$(&H0A)
150  CALL ibwrt (brd0%, wrt$)
```

QuickBASIC/BASIC

```
EOSV% = &H0A
.
.
.
sta% = ileos (brd0%, EOSV% + &H0800)
sta% = ilwrt (brd0%, "123" + chr$(&H0A), 4)
```

IBEOS (continued) IBEOS

C

```
v = XEOS | '\n';
ibeos (brd0, v);
ibwrt (brd0, "123\n", 4);
```

5. To send END with linefeeds and to terminate reads on linefeeds, change line 100 in Example 4.

BASICA/QuickBASIC/BASIC

```
:
:
100  v% = EOSV% + &H0C00
```

QuickBASIC/BASIC

```
sta% = ileos (brd0%, EOSV% + &H0C00)
```

C

```
v = REOS | XEOS | 0x0A;
ibeos (brd0, v);
```

IBEOT IBEOT

Purpose: Enable/disable END message on write operations.

Format:

BASICA

```
CALL ibeot (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibeot (ud%, v%)
```

or

```
ileot (ud%, v%)
```

C

```
short ibeot (short ud, short v)
```

`ud` specifies a device or an interface board. If `v` is non-zero, the END message is sent automatically with the last byte of each write operation. If `v` is zero, END is not automatically sent. `ibeot` is needed only to alter the value from the configuration setting. (In the default configuration, this feature is enabled).

The END message is the assertion of the GPIB EOI signal. If the automatic END termination message is enabled, it is not necessary to use the EOS character to identify the last byte of a data string. `ibeot` is used primarily to send variable length data.

The sending of END with the EOS character is determined by the `ibeos` function and is not affected by the `ibeot` function.

The assignment made by this function remains in effect until `ibeot` is called again, the `ibonl` or `ibfind` function is called, or the system is restarted.

When `ibeot` is called and an error does not occur, `iberr` is returned with a one if automatic END message was previously enabled, or with a zero if it was previously disabled.

IBEOT (continued) IBEOT

Device IBEOT Function

If `ud` specifies a device, the END termination message method that is selected is used on all device I/O write operations to that device.

Board IBEOT Function

If `ud` specifies an interface board, the END termination message method that is selected is used on all board I/O write operations, regardless of what device is written to.

Refer also to *IBEOS*.

Device Function Example:

Send the END message with the last byte of all subsequent writes to the device `plotter`.

BASICA/QuickBASIC/BASIC

```
100   plotter$ = "DEV5"
110   CALL ibfind (plotter$, plotter%)
120   v% = 1     ' Enable sending of EOI.
130   CALL ibeot (plotter%, v%)
140   REM  It is assumed that wrt$ contains
150   REM  the data to be written to the
160   REM  plotter.
170   CALL ibwrt (plotter%, wrt$)
```

QuickBASIC/BASIC

```
plotter% = ilfind ("DEV5")
sta% = ileot (plotter%, 1)
sta% = ilwrt (plotter%, wrt$, cnt%)
```

C

```
plotter = ibfind ("DEV5");
ibeot (plotter, 1);
ibwrt (plotter, wrt, cnt);
```

IBEOT (continued) IBEOT

Board Function Examples:

1. Stop sending END with the last byte.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = 0    ' Disable sending of EOI.
   110  CALL ibeot (brd0%, v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ileot (brd0%, 0)
   ```

   C

   ```
   ibeot (brd0, 0);
   ```

2. Send the END message with the last byte of all write operations.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = 1       ' Enable sending of EOI.
   110  CALL ibeot (brd0%, v%)
   120  REM  It is assumed that wrt$ contains
   130  REM  the data to be written and all
   140  REM  Listeners have been addressed.
   150  CALL ibwrt (brd0%, wrt$)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ileot (brd0%, 1)
   sta% = ilwrt (brd0%, wrt$, cnt%)
   ```

   C

   ```
   ibeot (brd0, 1);
   ibwrt (brd0, wrt, cnt);
   ```

IBEVENT

Purpose: Return the next event.

Format:

BASICA

```
CALL ibevent (ud%, event%)
```

QuickBASIC/BASIC

```
CALL ibevent (ud%, event%)
```

or

```
ilevent (ud%, event%)
```

C

```
short ibevent (short ud, unsigned short *event)
```

`ud` specifies an interface board. `event` stores the event code.

The `ibevent` function is used to determine which GPIB event (Device Clear or Device Trigger) occurred. It is normally called when the EVENT bit has been set in `ibsta`. The variable `event` is filled in with one of the following values:

0 = No events are in the event queue.

1 = A Device Clear message was received.

2 = A Device Trigger message was received.

Upon returning from this function, `ibcnt` contains the number of events that remain in the event queue.

IBEVENT (continued) IBEVENT

This function is typically used in Talker/Listener (T/L) applications, not Controller applications. Often a T/L application must determine the order in which Device Clear and Device Trigger messages are received by the interface board. The usual DCAS and DTAS bits of `ibsta` are not sufficient in determining the order of the events. When the event queue is enabled (by using the `ibconfig` function to enable the EVENT bit of `ibsta`), any time the driver receives a DCAS or DTAS message, the event is stored in the event queue of the board and the EVENT bit is set in `ibsta`. If any I/O is in progress, it is stopped with the error EABO. The application program then can call `ibevent` to determine which event or events occurred and can process those events. The event queue must be empty (the EVENT must not be in `ibsta`) before any more I/O can be started.

If the event queue fills up, a call to `ibevent` returns the ETAB error along with the oldest event in the queue.

Board Function Example:

BASICA/QuickBASIC/BASIC

```
100  mask% = &H4400      ' TIMO EVENT
110  CALL ibwait(brd0%, mask%)
120  IF ibsta% and EVENT then GOTO 140
130  STOP
140  ibevent(brd0%, event%)
150  REM  event% contains the event code.
```

QuickBASIC/BASIC

```
sta% = ilwait (brd0%, &H4400)
IF sta% and EVENT THEN
   sta% = ilevent(brd%, event%)
   ' event% contains the event code.
END IF
```

IBEVENT (continued) IBEVENT

C

```
ibwait(brd0, TIMO | EVENT);
if (ibsta & EVENT) {
   ibevent(brd0, &event);
   /* event contains the event code */
```

IBFIND IBFIND

Purpose: Open device and return the unit descriptor associated with the given name.

Format:

BASICA

```
CALL ibfind (udname$, ud%)
```

QuickBASIC/BASIC

```
CALL ibfind (udname$, ud%)
   or
ud% = ilfind (udname$)
```

C

```
short ibfind (char udname [])
```

`udname` is a string containing a default or configured device or board name. `ud` is a variable containing the unit descriptor returned by `ibfind`.

`ibfind` returns a number that is used in each function to identify the particular device or board that is used for that function. Calling `ibfind` is required to associate a variable name in the application program with a particular device or board name. The name used in the `udname` argument must match the default or configured device or board name. The number referred to throughout this manual as a unit descriptor is returned here in the variable `ud`.

Note: For board calls, the unit descriptor may be substituted with an integer board index of zero (0) or one (1). This feature allows any of the NI-488 board functions to be used compatibly with the NI-488.2 procedures described in Chapter 4, *NI-488.2 Software Characteristics and Routines*.

IBFIND (continued) IBFIND

`ibfind` performs the equivalent of `ibonl` to open the specified device or board and to initialize software parameters to their default configuration settings. Use a variable name close to the actual name of the device or board to simplify programming effort.

The unit descriptor is valid until `ibonl` is used to place that device or interface board offline.

If the `ibfind` call fails, a negative number is returned in place of the unit descriptor. The most probable reason for a failure is that the string argument passed into `ibfind` does not exactly match the default or configured device or board name.

Device Function Example:

Assign the unit descriptor of the device named `DEV4` (Device number 4) to `dvm`.

BASICA/QuickBASIC/BASIC

```
100  devname$ = "DEV4"   ' Device name
110                      ' assigned at
120                      ' configuration time.
130  CALL ibfind (devname$, dvm%)
140  IF dvm% < 0 GOTO 1000 ' ERROR ROUTINE
```

QuickBASIC/BASIC

```
dvm% = ilfind ("DEV4")
IF dvm% < 0 GOTO 1000      ' JUMP IF ERROR
```

C

```
if ((dvm = ibfind ("DEV4")) & ERR) error ();
```

IBFIND (continued) IBFIND

Board Function Examples:

Assign the unit descriptor of the board `"GPIB0"` to `brd0`.

BASICA/QuickBASIC/BASIC

```
100  udname$ = "GPIB0"    ' Board name assigned
110                       ' at configuration
120                       ' time.
130  CALL ibfind (udname$, brd0%)
140  IF brd0% < 0 GOTO 1000   ' ERROR ROUTINE
```

QuickBASIC/BASIC

```
brd0% = ilfind ("GPIB0")
IF brd0% < 0 GOTO 1000        ' ERROR ROUTINE
```

C

```
short brd0;

brd0 = ibfind ("GPIB0");
if (brd0 < 0) error ();
```

IBGTS IBGTS

Purpose: Go from Active Controller to Standby.

Format:

BASICA

```
CALL ibgts (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibgts (ud%, v%)
```

or

```
ilgts (ud%, v%)
```

C

short `ibgts` (short `ud`, short `v`)

`ud` specifies an interface board. If `v` is non-zero, the GPIB board shadow handshakes the data transfer as an Acceptor, and when the END message is detected, the GPIB board enters a Not Ready For Data (NRFD) handshake holdoff state on the GPIB. If `v` is zero, no shadow handshake or holdoff is done.

The `ibgts` function makes the GPIB board go to the Controller Standby state and to unassert the ATN signal if it initially is the Active Controller. `ibgts` permits the GPIB Controller board to go to standby and therefore allow transfers between GPIB devices to occur without its intervention.

If the shadow handshake option is activated, the GPIB board participates in data handshake as an Acceptor without actually reading the data. It monitors the transfers for the END message and holds off subsequent transfers. Through this mechanism, the GPIB board can take control synchronously on a subsequent operation such as `ibcmd` or `ibrpp`.

Before performing an `ibgts` with shadow handshake, the `ibeos` function should be called to establish the proper EOS character or to disable EOS detection.

IBGTS (continued) IBGTS

The ECIC error results if the GPIB board is not CIC.

Refer also to *IBCAC*.

In the examples that follow, GPIB commands and addresses are coded as printable ASCII characters.

Board Function Examples:

Turn the ATN line off after unaddressing all Listeners (UNL or ASCII ?), addressing a Talker at hex 46 (ASCII F) and addressing a Listener at hex 31 (ASCII 1) so that the Talker can send data messages.

BASICA/QuickBASIC/BASIC

```
100   cmd$ = "?F1"     ' UNL MTA1 MLA2
110   CALL ibcmd (brd0%,cmd$)
120   v% = 1         ' Listen in continuous mode.
130   CALL ibgts (brd0%,v%)
```

QuickBASIC/BASIC

```
ilcmd (brd0%,"?F1",3)
sta% = ilgts (brd0%,1)
```

C

```
ibcmd (brd0,"?F1",3);
ibgts (brd0,1);
```

IBIST IBIST

Purpose: Set or clear individual status bit for Parallel Polls.

Format:

BASICA

```
CALL ibist (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibist (ud%, v%)
```

or

```
ilist (ud%, v%)
```

C

```
short ibist (short ud, short v)
```

`ud` specifies an interface board. If `v` is non-zero, the individual status bit is set. If `v` is zero, the bit is cleared.

The `ibist` function is used when the GPIB board participates in a parallel poll that is conducted by another device that is the Active Controller. The Active Controller conducts a parallel poll by asserting the EOI signal to send the Identify (IDY) message. While this message is active, each device which has been configured to participate in the poll responds by asserting a predetermined GPIB data line either true or false, depending on the value of its local ist bit. The GPIB board, for example, can be assigned to drive the DIO3 data line true if ist=1 and false if ist=0; conversely, it can be assigned to drive DIO3 true if ist=0 and false if ist=1.

The relationship between the value of ist, the line that is driven, and the sense at which the line is driven is determined by the Parallel Poll Enable (PPE) message in effect for each device. The GPIB board is capable of receiving this message either locally, via the `ibppc` function, or remotely, via a command from the Active Controller. Once the PPE message is executed, the `ibist` function changes the sense at which the line is driven during the parallel poll, and in this fashion the GPIB board can convey a one-bit, device-dependent message to the Controller.

IBIST (continued) IBIST

When `ibist` is called and an error does not occur, the previous value of ist is stored in `iberr`.

Refer also to *IBPPC*.

Board Function Example:

1. Set the individual status bit.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = 1    ' Any non-zero value will do.
   110  CALL ibist (brd0%,v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilist (brd0%,1)
   ```

   C

   ```
   ibist (brd0,1);
   ```

2. Clear the individual status bit.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = 0
   110  CALL ibist (brd0%,v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilist (brd0%,0)
   ```

   C

   ```
   ibist (brd0,0);
   ```

IBLINES IBLINES

Purpose: Return the status of the GPIB control lines.

Format:

BASICA

```
CALL iblines (ud%, clines%)
```

QuickBASIC/BASIC

```
CALL iblines (ud%, clines%)
```

or

```
illines (ud%, clines%)
```

C

short `iblines` (short `ud`, unsigned short `*clines`)

`ud` is a board descriptor. A *valid* mask is returned along with the GPIB control line state information in `clines`. The low-order byte (bits 0 through 7) of `clines` contains a mask indicating the capability of the GPIB interface board to sense the status of each GPIB control line. The upper byte (bits 8 through 15) contains the GPIB control line state information. The pattern of each byte is as follows:

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
EOI	ATN	SRQ	REN	IFC	NRFD	NDAC	DAV

To determine if a GPIB control line is asserted, first check the appropriate bit in the lower byte to determine if the line can be monitored. If the bit can be monitored (indicated by a 1 in the appropriate bit position), then check the corresponding bit in the upper byte. If the bit is set (1), the corresponding control line is asserted. If the bit is clear (0), the control line is unasserted.

For `iblines` to return valid data, a *well-behaved* IEEE-488 bus must exist. A *well-behaved* IEEE-488 bus is a bus in which all attached devices are following the IEEE-488 specification.

IBLINES (continued) IBLINES

Device/Board Function Example:

Test for Remote Enable (REN):

BASICA/QuickBASIC/BASIC

```
100 CALL ibfind ("GPIB0", brd0%)
110 CALL iblines (brd0%, clines%)
120 REN% = clines% AND &H10
130 IF REN% <> &H10 THEN GOTO 800
140 REN% = clines% AND ((clines%/256) AND &H10)
150 IF REN% <> &H10 THEN GOTO 900
160 PRINT "REN is asserted!" : STOP
800 PRINT "GPIB board is unable to monitor REN."
810 STOP
900 PRINT "REN is not asserted!" : STOP
920 END
```

QuickBASIC/BASIC

```
brd0% = ilfind ("GPIB0")
IF brd0% AND EERR THEN GOTO ERROR
CALL iblines (brd0%, clines%)
IF ibsta% AND EERR THEN GOTO ERROR
REN% = clines% AND &H10
IF REN% <> &H10 THEN GOTO BRDERR
REN% = REN% AND (clines%/256)
IF REN% <> &H10 THEN GOTO UNASSERTED
PRINT "REN is asserted." : STOP
ERROR:
 PRINT "GPIB Handler error." : STOP
BRDERR:
 PRINT "GPIB board is unable to monitor REN!"
 STOP
UNASSERTED:
 PRINT "REN is not asserted." : STOP
END
```

IBLINES **(continued)** **IBLINES**

C

```
unsigned short clines;

if ((brd0 = ibfind ("GPIB0")) < 0) error();
if ((ibsta = iblines (brd0, &clines)) < 0)
   error();
if (!(clines & 0x10)) {
   printf("GPIB board can't monitor REN!");
   exit();
}
if (clines & 0x1000) {
   printf("REN is asserted.");
   exit();
}
printf("REN is not asserted.");
```

IBLN IBLN

Purpose: Check for the presence of a device on the bus.

Format:

BASICA

```
CALL ibln (ud%, pad%, sad%, listen%)
```

QuickBASIC/BASIC

```
CALL ibln (ud%, pad%, sad%, listen%)
```

or

```
illn (ud%, pad%, sad%, listen%)
```

C

```
short ibln (short ud, short pad, short sad, short *listen)
```

`ud` is a board or device descriptor. `pad` (legal values are 0 to 30) specifies the primary GPIB address of the device. `sad` (legal values are hex 60 to 7e, or `NO_SAD`, or `ALL_SAD`) specifies the secondary GPIB address of the device.

The function `ibln` returns a non-zero value in the variable `listen` if a Listener is at the specified GPIB address.

Notice that the `sad` parameter can be a value in hex 60 to 7e or one of the constants `NO_SAD` or `ALL_SAD`. You can test for a Listener using only GPIB primary addressing by making `sad=NO_SAD`, or you can test all secondary addresses associated with a single primary address (a total of 31 device addresses) when you set `sad=ALL_SAD`. In this case, `ibln` sends the primary address and all secondary addresses before waiting for NDAC to settle. If the listen flag is true, you must search only the 31 secondary addresses associated with a single primary address to locate the Listener.

IBLN (continued) IBLN

The two special constants that can be used in place of a secondary address are as follows:

```
NO_SAD = 0
ALL_SAD = -1
```

In BASICA, QuickBASIC, and BASIC, a period (`.`) replaces the underscore (_); thus, the constants read as follows:

```
NO.SAD
ALL.SAD
```

If `ud` specifies a device, `ibln` tests for a Listener on the board associated with the given device.

Refer also to *IBDEV* and *IBFIND*.

Device/Board Function Example:

Test for a GPIB Listener at `pad 2` and `sad 0x60`:

BASICA/QuickBASIC/BASIC

```
100  pad% = 2
105  sad% = &H60
110  CALL ibln (ud%,pad%,sad%,listen%)
115  if listen% = 0 THEN RETURN
120  REM  Error:  no device at this address
```

QuickBASIC/BASIC

```
sta% = illn (ud%, 2, &H60, listen%)
if listen% = 0 THEN RETURN
```

C

```
ibsta = ibln (ud, 2, 0x60, &listen);
if (!listen) {
/* No Listener found at this address */
}
```

IBLOC IBLOC

Purpose: Go to local.

Format:

BASICA

```
CALL ibloc (ud%)
```

QuickBASIC/BASIC

```
CALL ibloc (ud%)
   or
illoc (ud%)
```

C

short `ibloc` (short `ud`)

`ud` specifies a device or an interface board.

Unless the Remote Enable line has been unasserted with the `ibsre` function, all device functions automatically place the specified device in remote program mode. `ibloc` is used to move devices temporarily from a remote program mode to a local mode until the next device function is executed on that device.

Device IBLOC Function

`ibloc` places the device indicated in local mode by calling `ibcmd` to send the following command sequence:

1. Talk address of the access board
2. Secondary address of the access board, if necessary
3. Unlisten (UNL)
4. Listen address of the device
5. Secondary address of the device, if necessary

IBLOC (continued) IBLOC

6. Go To Local (GTL)

Other command bytes may be sent as necessary.

Board IBLOC Function

If `ud` specifies an interface board, the board is placed in a local state by sending the local Return To Local (RTL) message, if it is not locked in remote mode. The LOK bit of the status word indicates whether the board is in a lockout state. The `ibloc` function is used to simulate a front panel RTL switch if the computer is used as an instrument.

Device Function Example:

Return the device `dvm` to local state.

BASICA/QuickBASIC/BASIC

```
100  CALL ibloc (dvm%)
```

QuickBASIC/BASIC

```
sta% = illoc (dvm%)
```

C

```
ibloc (dvm);
```

Board Function Examples:

Return the interface board `brd0%` to local state.

BASICA/QuickBASIC/BASIC

```
100  CALL ibloc (brd0%)
```

QuickBASIC/BASIC

```
sta% = illoc (brd0%)
```

IBLOC (continued) IBLOC

C

```
ibloc (brd0);
```

IBONL IBONL

Purpose: Place the device or interface board online or offline.

Format:

BASICA

```
CALL ibonl (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibonl (ud%, v%)
```

or

```
ilonl (ud%, v%)
```

C

`short ibonl (short ud, short v)`

`ud` specifies a device or an interface board. If `v` is non-zero, the device or interface board is enabled for operation (online). If `v` is zero, it is reset (offline).

After a device or an interface board is taken offline, the handle (`ud`) is no longer valid. Before accessing the board or device again, you must re-execute an `ibfind` or `ibdev` call to open the board or device.

Calling `ibonl` with `v` non-zero restores the default configuration settings of a device or interface board.

IBONL (continued) IBONL

Device Function Example:

1. Disable the device `plotter`.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = 0
   110  CALL ibonl (plotter%,v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilonl (plotter%,0)
   ```

   C

   ```
   ibonl (plotter,0);
   ```

2. Enable the device `plotter` after taking it offline temporarily.

   BASICA/QuickBASIC/BASIC

   ```
   100  udname$ = "PLOTTER"
   110  CALL ibfind (udname$,plotter%)
   120  REM  ibfind automatically places the
   130  REM  device online.
   ```

   QuickBASIC/BASIC

   ```
   plotter% = ilfind ("PLOTTER")
   ```

   C

   ```
   plotter = ibfind ("PLOTTER");
   ```

IBONL (continued) IBONL

3. Restore default configuration settings of the device `plotter`.

   BASICA/QuickBASIC/BASIC

```
100  v% = 1
110  CALL ibonl (plotter%,v%)
```

   QuickBASIC/BASIC

```
sta% = ilonl (plotter%,1)
```

   C

```
ibonl (plotter,1);
```

Board Function Examples:

1. Disable the interface board `brd0`.

   BASICA/QuickBASIC/BASIC

```
100  v% = 0
110  CALL ibonl (brd0%,v%)
```

   QuickBASIC/BASIC

```
sta% = ilonl (brd0%,0)
```

   C

```
ibonl (brd0,0);
```

2. Enable the interface board `brd0`.

   BASICA/QuickBASIC/BASIC

```
100  udname$ = "GPIB0"
110  CALL ibfind (udname$,brd0%)
120  REM  ibfind automatically places board
130  REM  online.
```

IBONL (continued) IBONL

QuickBASIC/BASIC

```
brd0% = ibfind ("GPIB0")
```

C

```
brd0 = ibfind ("GPIB0");
```

3. Restore default configuration settings of the interface board `brd0`.

BASICA/QuickBASIC/BASIC

```
100  v% = 1
110  CALL ibonl (brd0%,v%)
```

QuickBASIC/BASIC

```
sta% = ilonl (brd0%,1)
```

C

```
ibonl (brd0,1);
```

IBPAD IBPAD

Purpose: Change Primary Address.

Format:

BASICA

```
CALL ibpad (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibpad (ud%, v%)
```

or

```
ilpad (ud%, v%)
```

C

short `ibpad` (short `ud`, short `v`)

`ud` specifies a device or an interface board. `v` specifies the primary GPIB address. `ibpad` is needed only to alter the configuration setting.

There are 31 valid GPIB addresses, ranging from 0 to hex 1E; that is, the lower five bits of `v` are significant and they must not all be ones. An EARG error results if the value of `v` is not in this range.

The assignment made by this function remains in effect until `ibpad` is called again, the `ibonl` or `ibfind` function is called, or the system is restarted.

When `ibpad` is called and an error does not occur, the previous primary address is stored in `iberr`.

IBPAD (continued) IBPAD

Device IBPAD Function

If `ud` specifies a device, `ibpad` determines the talk and listen addresses based on the value of `v`. A device listen address is formed by adding hex 20 to the primary address; the talk address is formed by adding hex 40 to the primary address. A primary address of hex 10 corresponds to a listen address of hex 30 and a talk address of hex 50. The actual GPIB address of any device is set within that device, either with hardware switches or a software program. Refer to the device documentation for instructions.

Board IBPAD Function

If `ud` specifies a board, `ibpad` programs the board to respond to the address indicated by `v`.

Refer also to *IBSAD*, *IBONL.*

Device Function Example:

Change the primary GPIB address of `plotter` to hex A.

BASICA/QuickBASIC/BASIC

```
100  v% = &HA
110  CALL ibpad (plotter%,v%)
```

QuickBASIC/BASIC

```
sta% = ilpad (plotter%,&HA)
```

C

```
ibpad (plotter,0xA);
```

IBPAD (continued) IBPAD

Board Function Examples:

Change the primary GPIB address of the board `brd0` to hex 7.

BASICA/QuickBASIC/BASIC

```
100  V% = &H7
110  CALL ibpad (brd0%,V%)
```

QuickBASIC/BASIC

```
sta% = ilpad (brd0%,&H7)
```

C

```
ibpad (brd0,0x7);
```

IBPCT IBPCT

Purpose: Pass Control.

Format:

BASICA

```
CALL ibpct (ud%)
```

QuickBASIC/BASIC

```
CALL ibpct (ud%)
```

or

```
ilpct (ud%)
```

C

short `ibpct` (short `ud`)

`ud` specifies a device.

The `ibpct` function passes CIC authority to the specified device from the access board assigned to that device. The board automatically goes to Controller Idle State (CIDS). The function assumes that the device has Controller capability.

`ibpct` calls the board `ibcmd` function to send the following commands:

- Unlisten
- Listen address of the access board
- Talk address of the device
- Secondary address of the device, if applicable
- Take Control (TCT)

Other command bytes may be sent as necessary.

Refer to *IBCMD* for additional information.

IBPCT (continued) IBPCT

Device Function Example:

Pass control to the device `ibmxt`.

BASICA/QuickBASIC/BASIC

```
100  CALL ibpct (ibmxt%)
```

QuickBASIC/BASIC

```
sta% = ilpct (ibmxt%)
```

C

```
ibpct (ibmxt);
```

IBPPC IBPPC

Purpose: Parallel Poll Configure.

Format:

BASICA

```
CALL ibppc (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibppc (ud%, v%)
```

or

```
ilppc (ud%, v%)
```

C

short `ibppc` (short `ud`, short `v`)

`ud` specifies a device or an interface board. `v` must be either a valid parallel poll enable/disable command or zero.

`ibppc` returns the previous value of `v` in `iberr` if an error does not occur.

Device IBPPC Function

If `ud` specifies a device, the `ibppc` function enables or disables the device from responding to parallel polls.

`ibppc` calls the board `ibcmd` function to send the following commands:

- Talk address of the access board
- Unlisten
- Listen address of the device
- Secondary address of the device, if applicable

IBPPC (continued) IBPPC

- Parallel Poll Configure (PPC)
- Parallel Poll Enable (PPE) or Disable (PPD)

Other command bytes are sent if necessary.

Each of the 16 PPE messages specifies the GPIB data line (DIO1 through DIO8) and sense (one or zero) that the device must use when responding to a parallel poll. The assigned message is interpreted by the device along with the current value of the individual status (ist) bit to determine if the selected line is driven true or false. For example, if the PPE=hex 64, DIO5 is driven true if ist=0 and false if ist=1, and if PPE=hex 68, DIO1 is driven true if ist=1 and false if ist=0. Any PPD message or zero value cancels the PPE message in effect. You must know which PPE and PPD messages are sent and determine what the responses indicate.

Board IBPPC Function

If `ud` specifies an interface board, the board responds to a parallel poll by setting its Local Poll Enable (LPE) message to `v`.

Refer also to *IBCMD* and *IBIST* for additional information.

Device Function Example:

1. Configure `dvm` to respond with data line DIO5 true (ist=0).

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = &H64
   110  CALL ibppc (dvm%,v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilppc (dvm%,&H64)
   ```

   C

   ```
   ibppc (dvm,0x64);
   ```

IBPPC (continued) IBPPC

2. Configure `dvm` to respond with data line DIO1 true (ist=1).

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = &H68
   110  CALL ibppc (dvm%,v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilppc (dvm%,&H68)
   ```

   C

   ```
   ibppc (dvm,0x68);
   ```

3. Cancel the parallel poll configuration of `dvm`.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = &H70
   110  CALL ibppc (dvm%,v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilppc (dvm%,&H70)
   ```

   C

   ```
   ibppc (dvm,0x70);
   ```

Board Function Examples:

Configure board `brd0` to respond with data line DIO5 true (ist=0).

BASICA/QuickBASIC/BASIC

```
100  v% = &H64
110  CALL ibppc (brd0%,v%)
```

QuickBASIC/BASIC

```
sta% = ilppc (brd0%,&H64)
```

C

```
ibppc (brd0,0x64);
```

IBRD IBRD

Purpose: Read data from a device to a string.

Format:

BASICA

```
CALL ibrd (ud%, rd$)
```

QuickBASIC/BASIC

```
CALL ibrd (ud%, rd$)
```

or

```
ilrd (ud%, rd$, cnt&)
```

C

short `ibrd` (short `ud`, char `rd`[], unsigned long `cnt`)

`ud` specifies a board or a device. `rd` is the storage buffer for data. In BASICA, `rd$` can be no more than 255 bytes. In QuickBASIC and BASIC, `rd$` can contain up to 32 kilobytes-1 (2^{15}-1 bytes). In C, `rd` can contain up to 4 gigabytes-1 (2^{32}-1 bytes).

`ibrd` terminates when one of the following events occurs:

- The allocated buffer becomes full.
- An error is detected.
- The time limit is exceeded.
- An END message is detected.
- An EOS character is detected (if this option is enabled).

Transfer count may be less than expected if any of these terminating events, except for the first event, occurs.

IBRD (continued) IBRD

When `ibrd` completes, `ibsta` holds the latest device status, `ibcntl` is the number of bytes read, `ibcnt` is the 16-bit representation of the number of bytes read, and, if the ERR bit in `ibsta` is set, `iberr` is the first error detected.

Device IBRD Function

If `ud` specifies a device, the device is addressed to talk and the access board is addressed to listen. Then the data is read from the device.

Board IBRD Function

If `ud` specifies an interface board, the `ibrd` function reads from a GPIB device that is assumed to already be properly addressed by the CIC. In addition to the termination conditions previously listed, a board `ibrd` function also terminates if a Device Clear (DCL) or Selected Device Clear (SDC) command is received from the CIC.

If the access board is Active Controller, the board is placed in Standby Controller state with ATN off even after the operation completes. If the access board is not Active Controller, `ibrd` commences immediately.

If the board is CIC, the `ibcmd` function must be used prior to `ibrd` to address a device to talk and the board to listen.

An EADR error results if the board is CIC but has not been addressed to listen with the `ibcmd` function. An EABO error results if, for any reason, `ibrd` does not complete within the time limit.

Device Function Example:

Read 100 bytes of data from a device.

BASICA/QuickBASIC/BASIC

```
100  brd% = 0 : pad% = 10 : sad% = 0
110  tmo% = 15 : eot% = 1 : eos% = 0
120  CALL ibdev(brd%,pad%,sad%,tmo%,eot%,eos%,dvm%)
130  rd$ = space$(100)
140  CALL ibrd (dvm%, rd$)
```

IBRD (continued) IBRD

QuickBASIC/BASIC

```
dvm% = ildev(0, 10, 0, 15, 1, 0)
rd$ = space$ (100)
sta% = ilrd (dvm%, rd$, 100)
```

C

```
short dvm;
char rd [100];

dvm = ibdev(0, 10, 0, 15, 1, 0);
ibrd (dvm, rd, 100);
```

Board Function Examples:

1. Read 100 bytes of data from a device at talk address hex 4C (ASCII L) (the listen address of the board is hex 20 or ASCII <space>).

   BASICA/QuickBASIC/BASIC

```
100  brd0$ = "GPIB0"
110  CALL ibfind (brd0$, brd0%)       'open board
120  cmd$ = "? L"                   ' UNL MLA TAD
130  CALL ibcmd (brd0%, cmd$)
140  rd$ = space$(100)
150  CALL ibrd (brd0%, rd$)
```

   QuickBASIC/BASIC

```
brd0% = ilfind ("gpib0")              'open board
sta% = ilcmd (brd0%, "? L", 3)    'UNL TAD MLA
rd$ = space$ (100)
sta% = ilrd (brd0%, rd$, 100)
```

IBRD (continued) IBRD

C

```
short brd0;
char rd [100];

brd0 = ibfind ("gpib0");          /* open board */
ibcmd (brd0, "?L ", 3);          /* UNL TAD MLA */
ibrd (brd0, rd, 100);
```

2. To terminate the read on an EOS character, see the *IBEOS Board Function Example*.

3. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

IBRDA IBRDA

Purpose: Read data asynchronously to string.

Format:

BASICA

```
CALL ibrda (ud%, rd$)
```

QuickBASIC/BASIC

```
CALL ibrda (ud%, rd$)
   or
ilrda (ud%, rd$, cnt&)
```

C

short `ibrda` (short `ud,` char `rd` [], unsigned long `cnt)`

`ud` specifies a device or an interface board. `rd` identifies the storage buffer for data. In BASICA, `rd$` can be only 255 bytes. In QuickBASIC and BASIC, `rd$` can contain up to 32 kilobytes-1 (2^{15}-1 bytes). In C, `rd` can contain up to 4 gigabytes-1 (2^{32}-1 bytes).

`ibrda` is used in place of `ibrd` when the application program must perform other functions while processing the GPIB I/O operation. `ibrda` returns immediately after starting the I/O operation.

The three asynchronous I/O calls (`ibcmda`, `ibrda`, and `ibwrta`) are designed to allow an application to perform other functions (non-GPIB functions) while processing the I/O. Once the asynchronous I/O call has been initiated, further GPIB calls involving the device or access board are not allowed until the I/O has completed and the GPIB software and the application have been resynchronized.

IBRDA (continued) IBRDA

Resynchronization can be accomplished by using one of the following three functions:

Note: Resynchronization is only successful if the `ibsta` returned contains CMPL.

- `ibwait` - The driver and application are synchronized. The wait mask is arbitrary.
- `ibstop` - The asynchronous I/O is canceled, and the driver and application are synchronized.
- `ibonl` - The asynchronous I/O is canceled, the interface has been reset, and the driver and application are synchronized.

Any other GPIB call involving the device or access board returns the EOIP error.

Device IBRDA Function

If `ud` specifies a device, the device is addressed to talk and the access board is addressed to listen. Then the data is read from the device. Other command bytes may be sent as necessary.

Board IBRDA Function

If `ud` specifies an interface board, the `ibrda` function attempts to read from a GPIB device that is assumed to be already properly addressed.

If the board is CIC, the `ibcmd` function must be called prior to `ibrda` to address the device to talk and the board to listen. Otherwise, the actual CIC must perform the addressing.

If the board is Active Controller, the board is first placed in Standby Controller state with ATN off even after the read operation completes. If the board is not the Active Controller, the read operation commences immediately.

IBRDA (continued) IBRDA

An EADR error results if the interface board is CIC but has not addressed itself to listen with the `ibcmd` function.

Device Function Example:

Read 56 bytes of data from `tape` while doing other tasks.

BASICA/QuickBASIC/BASIC

```
100  REM  Perform device read.
110  rd$ = space$(56)
120  CALL ibrda (tape%, rd$)
130  REM  Perform other processing here, then
140  REM  wait for I/O completion or a
150  REM  timeout.
160  mask% = &H4100  ' TIMO CMPL
170  CALL ibwait (tape%, mask%)
180  REM  ibsta% indicates how the read
190  REM  terminated: CMPL, END, TIMO, or ERR.
```

QuickBASIC/BASIC

```
rd$ = space$(56)
sta% = ilrda(tape%, rd$, 56)
'perform other processing here
sta% = ilwait(tape%, &H4100)  'TIMO CMPL
```

C

```
char rd[56];
ibrda (tape, rd, 56);
/*   Perform other processing here.              */
ibwait (tape, TIMO | CMPL);
```

IBRDA (continued) IBRDA

Board Function Examples:

1. Read 56 bytes of data from a device at talk address hex 4C (ASCII L) (the board listen address is hex 20 or ASCII <space>).

   BASICA/QuickBASIC/BASIC

```
100  REM  Perform addressing in preparation
110  REM  for board read.
120  cmd$ = "? L"          ' UNL MLA TAD
130  CALL ibcmd (brd0%, cmd$)
140  REM  Perform board read.
150  rd$ = space$(56)
160  CALL ibrda (brd0%, rd$)
170  REM  Perform other processing here, then
180  REM  wait for I/O completion or timeout.
190  mask% = &H4100        ' TIMO CMPL
200  CALL ibwait (brd0%, mask%)
210  REM  ibsta% indicates how the read
220  REM  terminated:  CMPL, END, TIMO, or ERR.
```

   QuickBASIC/BASIC

```
sta% = ilcmd (brd0%, "? L", 3)
rd$ = space$(56)
sta% = ilrda (brd0%, rd$, 56)
'perform other processing here
sta% = ilwait (brd0%, &H4100)
```

   C

```
char rd[56];
ibcmd (brd0, "? L", 3);   /*   UNL MLA TAD    */
ibrda (brd0, rd, 56);
/*  Perform other processing here.            */
ibwait (brd0, TIMO | CMPL);
```

2. To terminate the read on an EOS character, see the *IBEOS Board Function Example*.

IBRDA **(continued)** **IBRDA**

3. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

IBRDF IBRDF

Purpose: Read data from GPIB into file.

Format:

BASICA

```
CALL ibrdf (ud%, flname$)
```

QuickBASIC/BASIC

```
CALL ibrdf (ud%, flname$)
```

or

```
ilrdf (ud%, flname$)
```

C

short `ibrdf` (short `ud`, char `flname` [])

`ud` specifies a device or an interface board. `flname` is the filename under which the data is stored. `flname` may be up to 50 characters long, including a drive and path designation.

`ibrdf` automatically opens the file as a binary file (not as a character file). If the file does not exist, `ibrdf` creates it. On exit, `ibrdf` closes the file.

An EFSO error results if it is not possible to open, create, seek, write, or close the specified file.

The `ibrdf` function terminates on any of the following events:

- An error is detected.
- The time limit is exceeded.
- An END message is detected.
- An EOS character is detected (if this option is enabled).
- A Device Clear (DCL) or Selected Device Clear (SDC) command is received from another device which is the CIC.

IBRDF (continued) IBRDF

After termination, `ibcntl` is the number of bytes read. `ibcnt` is the 16-bit representation of the number of bytes read.

When the device `ibrdf` function returns, `ibsta` holds the latest device status, `ibcntl` is the number of data bytes read, `ibcnt` is the 16-bit representation of the number of bytes read, and if the ERR bit in `ibsta` is set, `iberr` is the first error detected.

Device IBRDF Function

If `ud` specifies a device, the same board functions as the device `ibrd` function are performed automatically. The `ibrdf` function terminates on similar conditions as `ibrd`.

Board IBRDF Function

If `ud` specifies an interface board, the board `ibrd` function reads from a GPIB device that is assumed to be already properly addressed.

An EADR error results if the board is CIC but has not been addressed to listen with the `ibcmd` function. An EABO error results if, for any reason, the read operation does not complete within the time limit. An EABO error also results if the device that is to talk is not addressed and/or the operation does not complete within the time limit for whatever reason.

Device Function Example:

Read data from the device `rdr%` into the file `RDGS` on disk drive `B`.

BASICA/QuickBASIC/BASIC

```
110  flname$ = "B:RDGS"
120  CALL ibrdf (rdr%, flname$)
130  REM  ibsta% and ibcnt% show the results of
140  REM  the read operation.
```

QuickBASIC/BASIC

```
sta% = ilrdf (rdr%, "B:RDGS")
```

IBRDF (continued) IBRDF

C

```
ibrdf (rdr, "B:RDGS");
```

Board Function Examples:

1. Read data from a device at talk address &H4C (ASCII L) to the file RDGS on the current disk drive and then unaddress everyone (the GPIB board listen address is hex 20 or ASCII <space>).

   BASICA/QuickBASIC/BASIC

   ```
   100  REM  Perform addressing in preparation
   110  REM  for board read.
   120  cmd$ = "?L "    ' UNL TAD MLA
   130  CALL ibcmd (brd0%, cmd$)
   140  REM  Perform board read.
   150  flname$ = "RDGS"
   160  CALL ibrdf (brd0%, flname$)
   170  REM  ibsta% and ibcnt% show the results of
   180  REM  the read operation.
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilcmd (brd0%, "?L ", 3)  'UNL TAD MLA
   sta% = ilrdf (brd0%, "RDGS")
   ```

   C

   ```
   ibcmd (brd0, "?L ", 3);
   ibrdf (brd0, "RDGS");
   ```

2. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

IBRDI IBRDI

Purpose: Read data to integer array.

Format:

BASICA

```
CALL ibrdi (ud%, iarr%(0), cnt%)
```

QuickBASIC/BASIC

```
CALL ibrdi (ud%, iarr%(), cnt&)
```

or

```
ilrdi (ud%, iarr%(), cnt&)
```

`ibrdi` is not available in C because it is not necessary. In C, the `ibrd` call is adequate for receiving data into any type of buffer. BASICA, QuickBASIC, and BASIC, which have more rigid typing rules, require a different call for array buffers than for strings.

`ud%` specifies a device or an interface board. `iarr%` is the integer array into which data is read. `cnt` specifies the maximum number of bytes to be read.

`ibrdi` is similar to the `ibrd` function, which reads data into a character string variable. `ibrdi` can read up to 64 kilobytes-1 (2^{16} - 1 bytes) of data. As the data is read, each byte pair is treated as an integer and stored in `iarr%`.

Unlike `ibrd`, `ibrdi` stores the data directly into an integer array. No integer conversion of the data is needed for arithmetic operations.

In QuickBASIC and BASIC, the array may also be of type single, double, or long; thus, if the device returns binary data that is compatible with any of these types, the data can be read directly into the array.

Refer to *IBRD* and to *BASICA/QuickBASIC/C NI-488 I/O Calls and Functions* at the beginning of this chapter.

IBRDI (continued) IBRDI

Device Function Example:

Read 512 bytes of data from `tape` and store in the integer array `rd`.

BASICA

```
100 brd% = 0 : pad% = 6 : sad% = 0
110 tmo% = 14 : eot% = 1 : eos% = 0
120 CALL ibdev(brd%,pad%,sad%,tmo%,eot%,eos%,tape%)
130 cnt% = 512
140 REM  Array size is equal to cnt% divided by 2
150 DIM rd% (256)
160 CALL ibrdi (tape%, rd%(0), cnt%)
```

QuickBASIC/BASIC

Replace line 160 with:

```
CALL ibrdi (tape%, rd%(), cnt%)
```

or

```
DIM rd% (256) 'integer array size = byte cnt/2
CALL ibdev(0,6,0,14,1,0,tape%) 'open device
sta% = ilrdi (tape%, rd%(), 512)
```

IBRDI (continued) IBRDI

Board Function Examples:

1. Read 56 bytes of data into the integer array `rd%` from a device at talk address hex 4C (ASCII L) (the GPIB board listen address is hex 20 or ASCII space).

   BASICA

   ```
   100  gpib$ = "GPIB0"
   110  CALL ibfind (gpib0$, brd0%)
   120  cmd$ = "? L"       ' UNL MLA TAD
   130  CALL ibcmd (brd0%, cmd$)
   140  cnt% = 56
   150  REM  Array size = cnt% divided by 2.
   160  DIM rd% (28)
   170  CALL ibrdi (brd0%, rd%(0), cnt%)
   180  REM  ibsta% shows how the read terminated:
   190  REM  on CMPL, END, TIMO, or ERR.
   200  REM  Data is stored in rd%().
   ```

   QuickBASIC/BASIC

   Replace line 170 with:

   ```
   CALL ibrdi (brd0%, rd%(), cnt%)
   ```

   or

   ```
   DIM rd%(28) 'byte count / 2
   brd0% = ilfind ("gpib0")
   sta% = ilcmd (brd0%, "? L", 3)       'UNL MLA TAD
   sta% = ilrdi (brd0%, rd%(), 56)
   ```

2. To terminate the read on an EOS character, see the *IBEOS Board Function Example*.

3. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

IBRDIA IBRDIA

Purpose: Read data asynchronously to integer array.

Format:

BASICA

```
CALL ibrdia (ud%, iarr%(0), cnt%)
```

QuickBASIC/BASIC

```
CALL ibrdia (ud%, iarr%(), cnt&)
```

or

```
ilrdia (ud%, iarr%(), cnt&)
```

`ibrdia` is not available in C because it is not necessary. In C, the `ibrda` call is adequate for receiving data into any type of buffer. BASICA, QuickBASIC, and BASIC, which have more rigid typing rules, require a different call for array buffers than for strings.

`ud` specifies a device or an interface board. `iarr%` is the integer array into which data is read. `cnt` specifies the maximum number of bytes to be read.

`ibrdia` is similar to the `ibrda` function, which reads data into a character string variable. `ibrdia` can read asynchronously up to 64 kilobytes (2^{16}-1 bytes) of data from `ud` and store them in `iarr`. As the data is read, each byte pair is treated as an integer and stored in `iarr`.

Unlike `ibrda`, `ibrdia` stores the data directly into an integer array. No integer conversion of the data is needed for arithmetic operations.

In QuickBASIC and BASIC, the array may also be of type single, double, or long; thus, if the device returns binary data that is compatible with any of these types, the data can be read directly into the array.

Note: For QuickBASIC and BASIC, do not pass dynamic arrays to the asynchronous functions `ibrdia` and `ibwrtia` because their location may change during an I/O operation.

Refer to *IBRDA* and to *BASICA/QuickBASIC/BASIC/C NI-488 I/O Calls and Functions* at the beginning of this chapter.

IBRDIA (continued) IBRDIA

Device Function Example:

Read 56 bytes of data into the integer array `rd` from `tape` while performing other processing.

BASICA

```
100  REM  Perform device read.
110  cnt% = 56
120  REM  Array size is equal to cnt% divided
130  REM  by 2.
140  DIM rd% (28)
150  CALL ibrdia (tape%,rd%(0),cnt%)
160  MASK% = &H4100  ' TIMO CMPL
170  REM  Perform other processing here, then
180  REM  wait for I/O completion or a
190  REM  timeout.
200  CALL ibwait (tape%, mask%)
210  REM  ibsta% shows how the read terminated:
220  REM  CMPL, END, TIMO, or ERR.
230  REM  If CMPL or ERR is not set, continue
240  REM  processing.
250  IF (ibsta% AND &H8100) = 0 GOTO 160
```

QuickBASIC/BASIC

Replace line 150 with:

```
CALL ibrdia (tape%, rd%(), cnt%)
```

or

```
DIM rd% (28) 'space to store 56 bytes
sta% = ilrdia (tape%, rd%(), 56)
again:
sta% = ilwait (tape%, &H4100)
IF (sta% AND &H8100) THEN GOTO again
```

IBRDIA (continued) IBRDIA

Board Function Examples:

1. Read 56 bytes of data into the integer array `rd%` from a device at talk address hex 4C (ASCII L) (the GPIB board listen address is hex 20 or ASCII space).

   BASICA

```
100  REM  Perform addressing in preparation
110  REM  for board read.
120  cmd$ = "? L"       ' UNL MLA TAD
140  CALL ibcmd (brd0%, cmd$)
150  REM  Perform board read.
160  cnt% = 56
170  REM  Array size is equal to cnt% divided
180  REM  by 2.
190  DIM rd% (28)
200  CALL ibrdia (brd0%, rd%(0), cnt%)
210  mask% = &H4100
220  REM  Perform other processing here, then
230  REM  wait for I/O completion or a
240  REM  timeout.
250  CALL ibwait (brd0%, mask%)
260  REM  ibsta% shows how the read terminated:
270  REM  CMPL, END, TIMO, or ERR.
280  REM  If CMPL or ERR are not set,
290  REM  continue processing.
300  IF (ibsta% AND &H8100) = 0 GOTO 200
310  REM  Properly terminate the asynchronous I/O
320  mask% = &H4100  '  TIMO CMPL
330  CALL ibwait (brd0%, mask%)
340  REM  Data is stored in RD%.
```

IBRDIA (continued) IBRDIA

QuickBASIC/BASIC

Replace line 200 with:

```
CALL ibrdia (brd0%, rd%(), cnt%)
```

or

```
sta% = ilcmd (brd0%, "? L", 3)          'UNL MLA TAD
DIM rd% (28)
sta% = ilrdia (brd0%, rd%(), 56)
again:
  'do other processing here...
  sta% = ilwait (brd0%, 0)
  IF (ibsta% AND &H8100) = 0 GOTO again 'ERR CMPL
  'The I/O is now complete (check ibsta for any
errors).
```

2. To terminate the read on an EOS character, see the *IBEOS Board Function Example*.

3. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

IBRPP IBRPP

Purpose: Conduct a Parallel Poll.

Format:

BASICA

```
CALL ibrpp (ud%, ppr%)
```

QuickBASIC/BASIC

```
CALL ibrpp (ud%, ppr%)
```

or

```
ilrpp (ud%, ppr%)
```

C

short `ibrpp` (short `ud`, char `*ppr`)

`ud` specifies a device or an interface board. `ppr` stores the parallel poll response.

Device IBRPP Function

If `ud` specifies a device, all devices on its GPIB are polled in parallel using the access board of that device. This is done by executing the board `ibrpp` function with the appropriate access board specified.

Board IBRPP Function

If `ud` specifies a board, the `ibrpp` function causes the identified board to conduct a parallel poll of previously configured devices by sending the IDY message (ATN and EOI both asserted) and reading the response from the GPIB data lines.

An ECIC error results if the GPIB board is not CIC. If the GPIB board is Standby Controller, it takes control and asserts ATN (becomes Active) prior to polling. It remains Active Controller afterward.

IBRPP (continued) IBRPP

When conducting a parallel poll on a bus that contains a bus extender, the extender is normally not able to return the remote poll response before the poll is finished. To allow bus extenders to operate transparently to the application, the driver can be configured to use a longer parallel poll duration. This duration can be set with `IBCONF`, or at runtime by calling the `ibconfig` function.

In the examples that follow, some of the GPIB commands and addresses are coded as printable ASCII characters. The simplest means of specifying values is to use printable ASCII characters to represent values. When possible, ASCII characters should be used. This is the simplest means of specifying the values. Refer to Appendix A, *Multiline Interface Messages*, for conversions of numeric values to ASCII characters.

Some commands relevant to parallel polls are shown in Table 5-7.

Table 5-7. Parallel Poll Commands

Command	Hex Value	Meaning
PPC	05	Parallel Poll Configure
PPU	15	Parallel Poll Unconfigure
PPE	60	Parallel Poll Enable
PPD	70	Parallel Poll Disable

Parallel poll constants are defined in the appropriate declaration file.

IBRPP (continued) IBRPP

Device Function Example:

Remotely configure the device `lcrmtr` to respond positively on DIO3 if its individual status bit is 1, and then parallel poll all configured devices.

BASICA/QuickBASIC/BASIC

```
100  v% = &H6A
110  CALL ibppc (lcrmtr%, v%)
120  CALL ibrpp (lcrmtr%, ppr%)
```

QuickBASIC/BASIC

```
sta% = ilppc (lcrmtr%, &H6A)
sta% = ilrpp (lcrmtr%, ppr%)
```

C

```
ibppc (lcrmtr, 0x6A);
ibrpp (lcrmtr, &ppr);
```

Board Function Examples:

1. Remotely configure the board `brd0` at listen address hex 23 (ASCII #) to respond positively on DIO3 if its individual status bit is 1, and then parallel poll all configured devices.

BASICA/QuickBASIC/BASIC

```
100  REM  Send UNL, TAD, LAD, PPC, PPE, and UNL.
110  cmd$ = "?@#" + chr$(&H05) + "j?"
120  CALL ibcmd (brd0%, cmd$)
130  CALL ibrpp (brd0%, ppr%)
```

QuickBASIC/BASIC

```
cmd$ = "?@#" + chr$(&H05) + "j?"  'UNL TAD LAD PPC PPE UNL
sta% = ilcmd (brd0%, cmd$, 6)
sta% = ilrpp (brd0%, ppr%)
```

IBRPP (continued) IBRPP

C

```
cmd[0] = UNL;
cmd[1] = 0x40
cmd[2] = 0x23;
cmd[3] = PPC;
cmd[4] = PPE | S | 2;
cmd[5] = UNL;
ibcmd (brd0, cmd, 6);
ibrpp (brd0, &ppr);
```

2. Disable and unconfigure all GPIB devices from parallel polling using the PPU (hex 15) command.

BASICA/QuickBASIC/BASIC

```
100  cmd$ = chr$(&H15)        ' PPU
110  CALL ibcmd (gpib0%, cmd$)
```

QuickBASIC/BASIC

```
sta% = ilcmd (gpib0%, chr$(&H15), 1)
```

C

```
ibcmd (gpib0, "\x15", 1);
```

IBRSC IBRSC

Purpose: Request or release system control.

Format:

BASICA

```
CALL ibrsc (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibrsc (ud%, v%)
```

or

```
ilrsc (ud%, v%)
```

C

short `ibrsc` (short `ud`, short `v`)

`ud` specifies an interface board. If `v` is non-zero, functions requiring System Controller capability are subsequently allowed. If `v` is zero, functions requiring System Controller capability are not allowed.

The `ibrsc` function is used to enable or disable the capability of the GPIB board to send the Interface Clear (IFC) and Remote Enable (REN) messages to GPIB devices using the `ibsic` and `ibsre` functions, respectively. The interface board must not be System Controller to respond to IFC sent by another Controller.

In most applications, the GPIB board is always the System Controller, but in some applications, the GPIB board is never the System Controller. In either case, the `ibrsc` function is used only if the computer is not going to be System Controller for the duration of the program execution. While the IEEE-488 standard does not specifically allow schemes in which System Control can be passed dynamically from one device to another, the `ibrsc` function can be used in such a scheme.

When `ibrsc` is called and an error does not occur, `iberr` is set to one if the interface board was previously System Controller and zero if it was not.

IBRSC (continued) IBRSC

Board Function Examples:

Request to be System Controller if the interface board `brd0` is not currently so designated.

BASICA/QuickBASIC/BASIC

```
100  v% = 1        ' Any non-zero value will do.
110  CALL ibrsc (brd0%, v%)
```

QuickBASIC/BASIC

```
sta% = ilrsc (brd0%, 1)
```

C

```
ibrsc (brd0, 1);
```

IBRSP IBRSP

Purpose: Return serial poll byte.

Format:

BASICA

```
CALL ibrsp (ud%, spr%)
```

QuickBASIC/BASIC

```
CALL ibrsp (ud%, spr%)
```

or

```
ilrsp (ud%, spr%)
```

C

short `ibrsp` (short `ud`, char `*spr`)

`ud` specifies a device. `spr` stores the serial poll response.

The `ibrsp` function is used to serial poll one device and obtain its status byte or to obtain a previously stored status byte. If bit 6 (the hex 40 bit) of the response is set, the device is requesting service.

When the automatic serial polling feature is enabled, the specified device may have been polled previously. If it has been polled and a positive response was obtained, the RQS bit of `ibsta` is set on that device. In this case, `ibrsp` returns the previously acquired status byte. If the RQS bit of `ibsta` is not set during an automatic poll, it serial polls the device.

When a poll is actually conducted, the specific sequence of events is as follows:

1. Unlisten (UNL)
2. Controllers Listen Address
3. Secondary address of the access board, if applicable

IBRSP (continued) IBRSP

4. Serial Poll Enable (SPE)
5. Talk address of the device
6. Secondary address of the device, if applicable
7. Read serial poll response byte from device
8. Serial Poll Disable (SPD)
9. Other command bytes may be sent as necessary

The response byte `spr`, except the RQS bit, is device specific. For example, the polled device might set a particular bit in the response byte to indicate that it has data to transfer and another bit to indicate a need for reprogramming. Consult the device documentation for interpretation of the response byte.

The IEEE-488.1 specification does not specify a length of time that the Controller should wait when reading the serial poll response byte from the device. The driver's default of 1 second is sufficient for most devices. If you need to make it longer, use `IBCONF` or the `ibconfig` function.

Refer to *IBCMD* and *IBRD* for additional information.

Device Function Example:

Obtain the serial poll response (`spr`) byte from the device `tape`.

BASICA/QuickBASIC/BASIC

```
100  CALL ibrsp (tape%, spr%)
110  REM  The application program would then
120  REM  analyze the response in spr%.
```

QuickBASIC/BASIC

```
sta% = ilrsp (tape%, spr%)
```

C

```
ibrsp (tape,&spr);
```

IBRSV IBRSV

Purpose: Request service and/or set or change the serial poll status byte.

Format:

BASICA

```
CALL ibrsv (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibrsv (ud%, v%)
```

or

```
ilrsv (ud%, v%)
```

C

```
short ibrsv (short ud, short v)
```

`ud` specifies an interface board. `v` is the status byte that the GPIB board provides when serial polled by another device that is the GPIB CIC. If bit 6 (the hex 40 bit) is set, the GPIB board additionally requests service from the Controller by asserting the GPIB SRQ line.

The `ibrsv` function is used to request service from the Controller using the Service Request (SRQ) signal and to provide a system-dependent status byte when the Controller serial polls the GPIB board.

When `ibrsv` is called and an error does not occur, the previous value of `v` is stored in `iberr`.

If the SPOLL bit of `ibsta` has been enabled by the `ibconfig` function, the SPOLL bit will be cleared from `ibsta` after a call to this function. If the bit 6 is set in the status byte (v), the SPOLL bit will be set in `ibsta` when the board is serial polled by the Controller.

IBRSV (continued) IBRSV

Board Function Examples:

1. Set the serial poll status byte to hex 41, which simultaneously requests service from an external CIC.

   BASICA/QuickBASIC/BASIC

   ```
   100  stb% = 1
   110  v% = &H40 OR stb%                    'assert SRQ
   120  CALL ibrsv (brd0%, v%)
   ```

   QuickBASIC/BASIC

   ```
   stb% = 1
   sta% = ilrsv (brd0%, stb% OR &H40)
   ```

   C

   ```
   stb% = 1
   ibrsv (brd0, stb | 0x41);
   ```

2. Change the status byte without requesting service.

   BASICA/QuickBASIC/BASIC

   ```
   100  stb% = &H23  ' New status byte value.
   110  CALL ibrsv (brd0%, stb%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilrsv (brd0%, &H23)
   ```

   C

   ```
   ibrsv (brd0, 0x23);
   ```

IBSAD IBSAD

Purpose: Change or disable Secondary Address.

Format:

BASICA

```
CALL ibsad (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibsad (ud%, v%)
```

or

```
ilsad (ud%, v%)
```

C

short `ibsad` (short `ud`, short `v`)

`ud` specifies a device or an interface board. If `v` is a number between hex 60 and hex 7E, that number becomes the secondary GPIB address device or interface board. If `v` is zero, secondary addressing is disabled. `ibsad` is needed only to alter the secondary address value from its configuration setting.

The assignment made by this function remains in effect until `ibsad` is called again, the `ibonl` or `ibfind` function is called, or the system is restarted.

When `ibsad` is called and an error does not occur, the previous secondary address is stored in `iberr`.

Device IBSAD Function

If `ud` specifies a device, the function enables or disables extended GPIB addressing for the device. When secondary addressing is enabled, the specified secondary GPIB address of that device is sent automatically in subsequent device I/O functions.

IBSAD (continued) IBSAD

Board IBSAD Function

If `ud` specifies an interface board, the `ibsad` function enables or disables extended GPIB addressing and, when enabled, assigns the secondary address of the GPIB board.

Refer also to *IBPAD* and *IBONL*.

Device Function Example:

1. Change the secondary GPIB address of `plotter` from its current value to hex 6A.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = &H6A
   110  CALL ibsad (plotter%, v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilsad (plotter%, &H6A)
   ```

   C

   ```
   ibsad (dvm, 0x6A);
   ```

2. Disable secondary addressing for the device `dvm`.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = 0
   110  CALL ibsad (dvm%,v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilsad (dvm%,0)
   ```

   C

   ```
   ibsad (dvm,0);
   ```

IBSAD (continued) IBSAD

Board Function Examples:

1. Change the secondary GPIB address of the interface board `brd0` from its current value to hex 6A.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = &H6A
   110  CALL ibsad (brd0%,v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilsad (brd0%,&H6A)
   ```

   C

   ```
   ibsad (brd0,0x6A);
   ```

2. Disable secondary addressing for the interface board `brd0`.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = 0
   110  CALL ibsad (brd0%,v%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilsad (brd0%,0)
   ```

   C

   ```
   ibsad (brd0,0);
   ```

IBSIC IBSIC

Purpose: Send interface clear for 100 µsec.

Format:

BASICA

```
CALL ibsic (ud%)
```

QuickBASIC/BASIC

```
CALL ibsic (ud%)
   or
ilsic (ud%)
```

C

short `ibsic` (short `ud`)

`ud` specifies an interface board. `ibsic` must be used at the beginning of a program if board functions are used.

The `ibsic` function asserts the IFC signal for at least 100 µsec if the GPIB board is System Controller. This action initializes the GPIB, makes the interface board CIC and Active Controller with ATN asserted, and is generally used when a bus fault condition is suspected.

The IFC signal resets only the GPIB interface functions of bus devices and not the internal device functions. Device functions are reset with the Device Clear (DCL) and Selected Device Clear (SDC) commands. To determine the effect of these messages, consult the device documentation.

The ESAC error occurs if the GPIB board does not have System Controller capability.

Refer also to *IBRSC*.

IBSIC (continued) IBSIC

Board Function Example:

At the beginning of a program, initialize the GPIB and become CIC and Active Controller.

BASICA/QuickBASIC/BASIC

```
100  CALL ibsic (brd0%)
```

QuickBASIC/BASIC

```
sta% = ilsic (brd0%)
```

C

```
ibsic (brd0);
```

IBSRE

Purpose: Set or clear the Remote Enable line.

Format:

BASICA

```
CALL ibsre (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibsre (ud%, v%)
```

or

```
ilsre (ud%, v%)
```

C

short `ibsre` (short `ud`, short `v`)

`ud` specifies an interface board. If `v` is non-zero, the Remote Enable (REN) signal is asserted. If `v%` is zero, the signal is unasserted.

The `ibsre` function turns the REN signal on and off. REN is used by devices to select between local and remote modes of operation. A device does not actually enter remote mode until it receives its listen address.

The ESAC error occurs if the GPIB board is not System Controller.

When `ibsre` is called and an error does not occur, the previous REN state is stored in `iberr`.

Refer also to *IBRSC*.

IBSRE (continued) IBSRE

Board Function Examples:

1. Place the device at listen address hex 23 (ASCII #) in remote mode.

   BASICA/QuickBASIC/BASIC

   ```
   100  v% = 1      ' Any non-zero value will do.
   110  CALL ibsre (brd0%,v%)
   120  cmd$ = "#"  ' LAD
   130  CALL ibcmd (brd0%,cmd$)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilsre (brd0%,1)
   sta% = ilcmd (brd0%,"#",1)
   ```

   C

   ```
   ibsre (brd0,1);
   ibcmd (brd0,"#",1);
   ```

2. To exclude the ability of the device to return to local mode, send the Local Lockout (LLO or hex 11) command or include it in the command string at 120 in Example 1.

   BASICA/QuickBASIC/BASIC

   ```
   140  cmd$ = chr$(&H11)
   150  CALL ibcmd (brd0%,cmd$)
   ```

   or

   ```
   120  cmd$ = "#" + chr$(&H11)
   130  CALL ibcmd (brd0%,cmd$)
   ```

IBSRE **(continued)** **IBSRE**

QuickBASIC/BASIC

```
sta% = ilcmd (brd0%,chr$ (&H11),1)
```

or

```
sta% = ilcmd (brd0%,"#" + chr$ (&H11),2)
```

C

```
ibcmd (brd0,"\x11",1);
```

or

```
ibcmd (brd0,"#\x11",2);
```

3. Return all devices to local mode.

BASICA/QuickBASIC/BASIC

```
100  v% = 0    ' Set REN to false.
110  CALL ibsre (brd0%, v%)
```

QuickBASIC/BASIC

```
sta% = ilsre (brd0%, 0)
```

C

```
ibsre (brd0, 0);
```

IBSRQ IBSRQ

Purpose: Register an SRQ "interrupt routine".

Format:

C

short `ibsrq` (void (far *`func`) (void))

This function establishes the C routine `func` as the procedure to be called whenever the driver notices the SRQI bit set (1) in the status word (`ibsta`) of a GPIB interface board. The check for SRQI is made after each call to the driver. If SRQI is set, `func` will be called before control is returned to the application program. SRQ servicing is turned off if `ibsrq` is called with `func` equal to `NULL`.

Note: Disable automatic serial polling if you use `ibsrq`.

For BASICA, QuickBASIC, and BASIC, use the ON PEN mechanism described in Chapter 3, *Understanding the NI-488.2 Software*, for SRQ servicing.

Note: If you use `ibsrq` or the ON PEN mechanism, you may need to disable your mouse driver. Many mouse drivers use the light pen interrupt.

IBSRQ (continued) IBSRQ

Example:

Establish `srqservice()` as the function to call for SRQ servicing.

C

```
short dvm;

void far srqservice() {

char spr;

ibrsp (dvm, &spr);

/* analyze the response here */
}

main () {

short gpib0 = ibfind ("gpib0");

/* disable autopolling */
ibconfig (gpib0, IbcAUTOPOLL, 0);

dvm = ibfind ("DEV4");
ibsrq (srqservice);

}
```

IBSTOP IBSTOP

Purpose: Abort asynchronous operation.

Format:

BASICA

```
CALL ibstop (ud%)
```

QuickBASIC/BASIC

```
CALL ibstop (ud%)
   or
ilstop (ud%)
```

C

```
short ibstop (short ud)
```

`ud` specifies a device or an interface board.

`ibstop` terminates any asynchronous read, write, or command operation and then resynchronizes the application with the driver.

If there is an asynchronous I/O operation in progress, the ERR bit in the status word is set and an EABO error is returned.

Device IBSTOP Function

If `ud` specifies a device, `ibstop` attempts to terminate any unfinished asynchronous I/O device function to that device.

Board IBSTOP Function

If `ud` specifies a board, `ibstop` attempts to terminate any unfinished asynchronous I/O operation that had been started with that board.

IBSTOP (continued) IBSTOP

Device Function Example:

Stop any asynchronous operation associated with the device `rdr`.

BASICA/QuickBASIC/BASIC

```
100  CALL ibstop (rdr%)
```

QuickBASIC/BASIC

```
sta% = ilstop (rdr%)
```

C

```
ibstop (rdr);
```

Board Function Examples:

Stop any asynchronous operation associated with the board `brd0`.

BASICA/QuickBASIC/BASIC

```
100  CALL ibstop (brd0%)
```

QuickBASIC/BASIC

```
sta% = ilstop (brd0%)
```

C

```
ibstop (brd0);
```

IBTMO IBTMO

Purpose: Change or disable time limit.

Format:

BASICA

```
CALL ibtmo (ud%, v%)
```

QuickBASIC/BASIC

```
CALL ibtmo (ud%, v%)
   or
iltmo (ud%, v%)
```

C

short `ibtmo` (short `ud`, short `v`)

`ud` specifies a device or an interface board. `v` specifies the time limit as follows:

Table 5-8. Timeout Code Values

Value of v	**Minimum Timeout**
0	disabled
1	10 µsec
2	30 µsec
3	100 µsec
4	300 µsec

(continues)

IBTMO (continued) IBTMO

Table 5-8. Timeout Code Values (continued)

Value of v	**Minimum Timeout**
5	1 msec
6	3 msec
7	10 msec
8	30 msec
9	100 msec
10	300 msec
11	1 sec
12	3 sec
13	10 sec
14	30 sec
15	100 sec

Note: If `v` is zero, no limit is in effect.

`ibtmo` is needed only to alter the value from its configuration setting.

The assignment made by this function remains in effect until `ibtmo` is called again, the `ibonl` or `ibfind` function is called, or the system is restarted.

IBTMO (continued) IBTMO

The `ibtmo` function changes the length of time that many functions wait for an I/O operation to finish. These functions include most functions that access the GPIB bus. Some of these functions are as follows:

- `ibcmd`
- `ibrd`
- `ibrdi`
- `ibwrt`
- `ibwrti`

The `ibtmo` function also changes the length of time that device functions wait for commands to be accepted. If a device does not accept commands within the time limit, the EBUS error is returned.

When `ibtmo` is called and an error does not occur, the previous timeout code value is stored in `iberr`.

Device IBTMO Function

If `ud` specifies a device, the new time limit is used in subsequent device functions directed to that device.

Board IBTMO Function

If `ud` specifies a board, the new time limit is used in subsequent board functions directed to that board.

Refer also to *IBWAIT*.

IBTMO (continued) IBTMO

Device Function Example:

Change the time limit for calls involving the device `tape` to approximately 300 msec.

BASICA/QuickBASIC/BASIC

```
100  tape$ = "DEV9"
110  CALL ibfind(tape$, tape%)
120  v% = 10
130  CALL ibtmo (tape%, v%)
```

QuickBASIC/BASIC

```
tape% = ilfind ("DEV9")
sta% = iltmo (tape%,10)
```

C

```
tape = ibfind ("dev9");
ibtmo (tape, 10);
```

Board Function Examples:

Change the time limit to 10 msec for board functions using `brd0`.

BASICA/QuickBASIC/BASIC

```
100  v% = 7
110  CALL ibtmo (brd0%, v%)
```

QuickBASIC/BASIC

```
sta% = iltmo (brd0%, 7)
```

C

```
ibtmo (brd0,7);
```

IBTRAP IBTRAP

Purpose: Alter trap and display modes of the Applications Monitor.

Format:

BASICA

```
CALL ibtrap (mask%, v%)
```

QuickBASIC/BASIC

```
CALL ibtrap (mask%, v%)
   or
iltrap (mask%, v%)
```

C

```
short ibtrap (short mask, short mode)
```

`mask` specifies a bit mask with the same bit assignments as `ibsta`. Each `mask` bit is set to be trapped and/or recorded (depending on the value of `mode`) when the corresponding bit appears in the status word after a GPIB call. If all the bits are set, then every GPIB call is trapped.

`mode` determines whether the recording and trapping occur. The valid values are listed in Table 5-9.

Table 5-9. IBTRAP Modes

Value	Effect
1	Turn monitor off. No recording or trapping occurs.
2	Turn record on. All calls are recorded but no trapping occurs.
3	Turn record and trap on. All calls are recorded and the monitor is displayed whenever a trap condition occurs.

IBTRAP (continued) IBTRAP

If an error occurs during a call to `ibtrap`, the ERR bit of `ibsta` is set and `iberr` is one of the values listed in Table 5-10.

Table 5-10. IBTRAP Errors

Value	Explanation
ECAP	Applications monitor is not installed.
EARG	Invalid monitor mode.

Otherwise, `iberr` contains the previous `mask` value.

Refer to Chapter 7, *Applications Monitor*, for more information.

Device Function Example:

Configure applications monitor to record and trap on SRQ or CMPL.

BASICA/QuickBASIC/BASIC

```
100  mode% = 3              'Record and trap on
110  mask% = &H1100         'SRQ or CMPL
120  CALL ibtrap (mask%, mode%)
```

QuickBASIC/BASIC

```
sta% = iltrap (&H1100, 3)
```

C

```
ibtrap (0x1100, 3);
```

IBTRG IBTRG

Purpose: Trigger selected device.

Format:

BASICA

```
CALL ibtrg (ud%)
```

QuickBASIC/BASIC

```
CALL ibtrg (ud%)
```
or
```
iltrg (ud%)
```

C

short `ibtrg` (short `ud`)

`ud` specifies a device.

`ibtrg` addresses and triggers the specified device.

`ibtrg` sends the following commands:

- Talk address of access board
- Secondary address of access board, if applicable
- Unlisten
- Listen address of the device
- Secondary address of the device, if applicable
- Group Execute Trigger (GET)

Other command bytes may be sent as necessary.

Refer to *IBCMD* for additional information.

IBTRG (continued) IBTRG

Device Function Example:

Trigger the device `analyz`.

BASICA/QuickBASIC/BASIC

```
100  CALL ibtrg (analyz%)
```

QuickBASIC/BASIC

```
sta% = iltrg (analyz%)
```

C

```
ibtrg (analyz);
```

IBWAIT IBWAIT

Purpose: Wait for selected event.

Format:

BASICA

```
CALL ibwait (ud%, mask%)
```

QuickBASIC/BASIC

```
CALL ibwait (ud%, mask%)
```

or

```
ilwait (ud%, mask%)
```

C

short `ibwait` (short `ud`, short `mask`)

`ud` specifies a device or an interface board. `mask` is a bit mask with the same bit assignments as the status word, `ibsta`. `ibwait` is used to monitor the events selected by the bits in `mask` and to delay processing until any of them occur. These events and bit assignments are shown in Table 5-11.

The declaration file for BASICA/QuickBASIC/BASIC/C defines the mnemonic for each bit in the status bytes `ibsta` and `iberr`. For example, if you are programming in QuickBASIC, the following two calls are equivalent:

- `IF IBSTA% AND TACS THEN PRINT "TALK ADDRESS"`
- `IF IBSTA% AND &H0008 THEN PRINT "TALK ADDRESS"`

IBWAIT (continued) IBWAIT

Table 5-11. Wait Mask Layout

Mnemonic	Bit Pos.	Hex Value	Description
ERR	15	8000	GPIB error
TIMO	14	4000	Time limit exceeded
END	13	2000	GPIB board detected END or EOS
SRQI	12	1000	SRQ on
RQS	11	800	Device requesting service
EVENT	10	400	A DTAS or DCAS event has occurred
SPOLL	9	200	The board has been serial polled by the Controller
CMPL	8	100	Asynchronous I/O completed
LOK	7	80	GPIB board is in lockout state
REM	6	40	GPIB board is in remote state
CIC	5	20	GPIB board is CIC
ATN	4	10	Attention is asserted
TACS	3	8	GPIB board is Talker
LACS	2	4	GPIB board is Listener
DTAS	1	2	GPIB board is in device trigger state
DCAS	0	1	GPIB board is in device clear state

IBWAIT (continued) IBWAIT

`ibwait` also updates `ibsta`. If `mask`=0 or `mask`=hex 8000 (the ERR bit), the function returns immediately.

If the TIMO bit is zero or the time limit is set to zero with the `ibtmo` function, timeouts are disabled. Disabling timeouts should be done only when setting `mask`=0 or when it is certain the selected event will occur; otherwise, the processor may wait indefinitely for the event to occur.

Device IBWAIT Function

If `ud` specifies a device, only the ERR, TIMO, END, RQS, and CMPL bits of the wait mask and status word are applicable. If automatic polling is enabled, then on an `ibwait` for RQS, each time the GPIB SRQ line is asserted, the access board of the specified device serial polls all devices on its GPIB and saves the responses, until the status byte returned by the device being waited for indicates that it was the device requesting service (bit hex 40 is set in the status byte). If the TIMO bit is set, `ibwait` returns if the event does not occur within the timeout period of the device.

Board IBWAIT Function

If `ud` specifies a board, all bits of the wait mask and status word are applicable except RQS.

Device Function Example:

Wait indefinitely for the device `logger` to request service.

BASICA/QuickBASIC/BASIC

```
100  mask% = &H800     ' RQS
110  CALL ibwait (logger%,mask%)
```

QuickBASIC/BASIC

```
sta% = ilwait (logger%,&H800)
```

C

```
mask = RQS;      /*   mask = 0x800;              */
ibwait (logger,mask);
```

IBWAIT (continued) IBWAIT

Board Function Examples:

1. Wait for a service request or a timeout.

   BASICA/QuickBASIC/BASIC

   ```
   100  mask% = &H5000    ' TIMO SRQI
   110  CALL ibwait (brd0%, mask%)
   120  REM  ibsta% indicates what occurred.
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilwait (brd0%, &H5000)
   ```

   C

   ```
   mask = SRQI | TIMO;      /*   mask = 0x5000;   */
   ibwait (brd0,mask);
   ```

2. Update the current status for `ibsta`.

   BASICA/QuickBASIC/BASIC

   ```
   100  mask% = 0
   110  CALL ibwait (brd0%,mask%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilwait (brd0%,0)
   ```

   C

   ```
   ibwait (ud,0);
   ```

3. Wait indefinitely until control is passed from another CIC.

   BASICA/QuickBASIC/BASIC

   ```
   100  mask% = &H20 ' CIC
   110  CALL ibwait (brd0%,mask%)
   ```

IBWAIT (continued) IBWAIT

QuickBASIC/BASIC

```
sta% = ilwait (brd0%,&H20)
```

C

```
mask = CIC;     /*   CIC = 0x20;                */
ibwait (ud,mask);
```

4. Wait indefinitely until addressed to talk or listen by another CIC.

BASICA/QuickBASIC/BASIC

```
100  mask% = &H0C ' TACS LACS
110  CALL ibwait (brd0%,mask%)
```

QuickBASIC/BASIC

```
sta% = ilwait (brd0%,&H0C)
```

C

```
mask = TACS | LACS;  /*  TACS | LACS = 0x0C;  */
ibwait (ud,mask);
```

IBWRT IBWRT

Purpose: Write data from string.

Format:

BASICA

```
CALL ibwrt (ud%, wrt$)
```

QuickBASIC/BASIC

```
CALL ibwrt (ud%, wrt$)
```

or

```
ilwrt (ud%, wrt$, cnt&)
```

C

int `ibwrt` (int `ud,` char `wrt` [], unsigned long `cnt`)

`ud` specifies a device or an interface board. `wrt` the buffer of data to be sent over the GPIB.

In BASICA, `wrt` can be only 255 bytes. In QuickBASIC and BASIC, `wrt` can contain up to 32 kilobytes (2^{15}-1 bytes). In C, `wrt` can contain up to 4 gigabytes-1 (2^{32} - 1 bytes).

The `ibwrt` terminates on any of the following events:

- All bytes are transferred.
- An error is detected.
- The time limit is exceeded.
- A Device Clear (DCL) or Selected Device Clear (SDC) command is received from another device which is the CIC.

After termination, `ibcntl` is the number of bytes read. `ibcnt` is the 16-bit representation of the number of bytes read. A short count can occur on any of the above terminating events but the first.

IBWRT (continued) IBWRT

When the device `ibwrt` function returns, `ibsta` holds the latest device status, `ibcntl` is the actual number of data bytes written to the device, `ibcnt` is the 16-bit representation of the number of data bytes written, and, if the ERR bit in `ibsta` is set, `iberr` is the first error detected.

Device IBWRT Function

If `ud` specifies a device, the device is addressed to listen and the access board is addressed to talk.

Then the data is written to the device.

Board IBWRT Function

If `ud` specifies an interface board, the `ibwrt` function attempts to write to a GPIB device that is assumed to be already addressed by the CIC.

If the access board is CIC, `ibcmd` must be called prior to `ibwrt` to address the device to listen and the board to talk.

If the access board is Active Controller, the board is first placed in Standby Controller state with ATN off even after the write operation completes. If the access board is not the Active Controller, `ibwrt` commences immediately.

An EADR error results if the board is CIC but has not been addressed to talk with `ibcmd`. An EABO error results if, for any reason, `ibwrt` does not complete within the time limit. An ENOL error occurs if there are no Listeners on the bus when the data bytes are sent.

Note: If you want to send an EOS character at the end of your data string, you must place it there explicitly. See *Device Function Example 2*.

IBWRT (continued) IBWRT

Device Function Example:

1. Write ten instruction bytes to the device `dvm`.

   BASICA/QuickBASIC/BASIC

   ```
   100  wrt$ = "F3R1X5P2G0"
   110  CALL ibwrt (dvm%, wrt$)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilwrt (dvm%, "F3R1X5P2G0", 10)
   ```

   C

   ```
   ibwrt (dvm, "F3R1X5P2G0", 10);
   ```

2. Write five instruction bytes terminated by a carriage return and a linefeed to the device `ptr`. Linefeed is the EOS character of the device.

   BASICA/QuickBASIC/BASIC

   ```
   100  wrt$ = "IP2X5" + chr$(&H0D) + chr$(&H0A)
   110  CALL ibwrt (ptr%, wrt$)
   ```

   QuickBASIC/BASIC

   ```
   wrt$ = "IP2X5" + chr$ (&H0D) + chr$ (&H0A)
   sta% = ilwrt (ptr%, wrt$, 7)
   ```

   C

   ```
   ibwrt (ptr, "IP2X5\r\n", 7);
   ```

IBWRT (continued) IBWRT

Board Function Examples:

1. Write ten instruction bytes to a device at listen address hex 2F (ASCII `/`) (GPIB board talk address is hex 40 (ASCII `@`)).

   BASICA/QuickBASIC/BASIC

   ```
   100  REM  Perform addressing.
   110  cmd$ = "?@/" ' UNL MTA LAD
   120  CALL ibcmd (brd0%,cmd$)
   130  REM  Perform board write.
   140  wrt$ = "F3R1X5P2G0"
   150  CALL ibwrt (brd0%,wrt$)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilcmd (brd0%"?@/",3)
   sta% = ilwrt (brd0%,"F3R1X5P2G0",10)
   ```

   C

   ```
   ibcmd (brd0,"?@/",3);
   ibwrt (brd0,"F3R1X5P2G0",10);
   ```

2. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

IBWRTA IBWRTA

Purpose: Write data asynchronously from string.

Format:

BASICA

```
CALL ibwrta (ud%,wrt$)
```

QuickBASIC/BASIC

```
CALL ibwrta (ud%, wrt$)
   or
ilwrta (ud%, wrt$, cnt&)
```

C

```
int ibwrta (int ud, char wrt [], unsigned long cnt)
```

`ud` specifies a device or an interface board. `wrt` contains the data to be sent over the GPIB.

In BASICA, `wrt` can only be 255 bytes. In QuickBASIC and BASIC, `wrt` can contain up to 32 kilobytes (2^{15}-1 bytes). In C, `wrt` can contain up to 4 gigabytes-1 (2^{32}-1 bytes).

`ibwrta` is used in place of `ibwrt` when the application program must perform other functions while processing the GPIB I/O operation. `ibwrta` returns immediately after starting the I/O operation.

The three asynchronous I/O calls (`ibcmda`, `ibrda`, and `ibwrta`) are designed to allow an application to perform other functions (non-GPIB functions) while processing the I/O. Once the asynchronous I/O call has been initiated, further GPIB calls involving the device or access board are not allowed until the I/O has completed and the GPIB driver and the application have been resynchronized.

IBWRTA (continued) IBWRTA

Resynchronization can be accomplished by using one of the following three functions:

Note: Resynchronization is only successful if the `ibsta` returned contains CMPL.

• `ibwait`	- The driver and application are synchronized. The wait mask is arbitrary.
• `ibstop`	- The asynchronous I/O is canceled, and the driver and application are synchronized.
• `ibonl`	- The asynchronous I/O is canceled, the interface has been reset, and the driver and application are synchronized.

Any other GPIB call involving the device or access board returns the EOIP error.

Device IBWRTA Function

If `ud` specifies a device, the device is addressed to listen and the access board is addressed to talk. Then the data is written to the device.

Board IBWRTA Function

If `ud` specifies an interface board, the `ibwrta` function attempts to write to a GPIB device that is assumed to be already properly initialized and addressed by the actual CIC.

If the board is CIC, the `ibcmd` function must be called prior to `ibwrta` to address the device to listen and the board to talk.

If the board is Active Controller, the board is first placed in Standby Controller state with ATN off (even after the write operation completes). Otherwise, the write operation commences immediately.

IBWRTA (continued) IBWRTA

An EADR error results if the board is CIC but has not been addressed to talk with the `ibcmd` function. The ENOL error does *not* occur if there are no Listeners.

Note: If you want to send an EOS character at the end of your data string, you must place it there explicitly.

When the device `ibwrt` function returns, `ibsta` holds the latest device status, and, if the ERR bit in `ibsta` is set, `iberr` is the first error detected.

Device Function Example:

Write ten instruction bytes to `dvm` while performing other processing.

BASICA/QuickBASIC/BASIC

```
100  wrt$ = "F3R1X5P2G0"
110  CALL ibwrta (dvm%,wrt$)
120  mask% = &H4100    'TIMO CMPL
130  REM  Perform other processing here, then
140  REM  wait for I/O completion or a
150  REM  timeout.
160  CALL ibwait (dvm%,mask%)
170  REM  ibsta% indicates how the write
180  REM  terminated:CMPL, END, TIMO, or ERR.
```

QuickBASIC/BASIC

```
ilwrta (dvm%,"F3RIX5P2G0",10)
sta% = ilwait (dvm%,&H4100)
```

C

```
ibwrta (dvm, "F3R1X5P2G0", 10);
/*   Perform other processing here               */
ibwait (dvm, TIMO | CMPL);
```

IBWRTA (continued) IBWRTA

Board Function Examples:

1. Write ten instruction bytes to a device at listen address hex 2F (ASCII /), while testing for a high priority event to occur and then unaddress it (the GPIB board talk address is hex 40 or ASCII @).

   BASICA/QuickBASIC/BASIC

   ```
   100  REM  Perform addressing in preparation
   110  REM  for board write.
   120  cmd$ = "?@/"      ' UNL MTA LAD
   130  CALL ibcmd (brd0%,cmd$)
   140  REM  Perform board asynchronous write.
   150  wrt$ = "F3R1X5P2G0"
   160  CALL ibwrta (brd0%,wrt$)
   170  REM  Perform other processing here, then
   180  REM  wait for I/O completion or timeout.
   190  mask% = &H4100                ' TIMO CMPL
   200  CALL ibwait (brd0%,mask%)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilcmd (brd0%,"?@/",3)
   sta% = ilwrta (brd0%,"F3RIX5P2G0",10)
   sta% = ilwait (brd0%,&H4100)
   ```

   C

   ```
   ibcmd (brd0,"?@/",3);    /*   UNL MTA LAD      */
   ibwrta (brd0,"F3R1X5P2G0",10);
   /* Perform other processing here.              */
   ibwait (brd0,TIMO | CMPL);
   ```

2. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

IBWRTF IBWRTF

Purpose: Write data from file.

Format:

BASICA

```
CALL ibwrtf (ud%, flname$)
```

QuickBASIC/BASIC

```
CALL ibwrtf (ud%, flname$)
```

or

```
ilwrtf (ud%, flname$)
```

C

```
int ibwrtf (int ud, char flname [])
```

`ud` specifies a device or an interface board. `flname` is the filename from which the data is written. `flname` may be up to 50 characters long, including a drive and path designation.

`ibwrtf` automatically opens the file. On exit, `ibwrtf` closes the file.

An EFSO error results if it is not possible to open, seek, read, or close the specified file.

The `ibwrtf` function operation terminates on any of the following events:

- All bytes sent.
- An error is detected.
- The time limit is exceeded.
- A Device Clear (DCL) or Selected Device Clear (SDC) command is received from another device that is the CIC.

After termination, `ibcntl` is the number of bytes written. `ibcnt` is the 16-bit representation of the number of bytes written.

IBWRTF (continued) IBWRTF

Device IBWRTF Function

If ud specifies a device, the same board functions as the device ibwrt function are performed automatically. It terminates on similar conditions as ibwrt.

When the ibwrtf function returns, ibsta holds the latest device status, ibcntl is the number of data bytes written, ibcnt is the 16-bit representation of the number of bytes written, and, if the ERR bit in ibsta is set, iberr is the first error detected.

Board IBWRTF Function

If ud specifies an interface board, the board ibwrt function writes to a GPIB device that is assumed to be already properly addressed.

An EADR error results if the board is CIC but has not been addressed to talk with the ibcmd function. An EABO error results if, for any reason, the read operation does not complete within the time limit. An ENOL error occurs if there are no Listeners on the bus when the data bytes are sent.

Device Function Example:

Write data to the device rdr from the file Y.DAT on the current drive.

BASICA/QuickBASIC/BASIC

```
100  flname$ = "Y.DAT"
110  CALL ibwrtf (rdr%,flname$)
```

QuickBASIC/BASIC

```
sta% = ilwrtf (rdr%,"Y.DAT")
```

C

```
ibwrtf (rdr,"Y.DAT");
```

IBWRTF (continued) IBWRTF

Board Function Examples:

1. Write data to the device at listen address hex 2C (ASCII `,`) from the file `Y.DAT` on the current drive, and then unaddress `brd0`.

   BASICA/QuickBASIC/BASIC

   ```
   100  REM  Perform addressing in preparation
   110  REM  for board write.
   120  cmd$ = "?@,"      ' UNL MTA LAD
   130  CALL ibcmd (brd0%,cmd$)
   140  REM  Perform board write.
   150  flname$ = "Y.DAT"
   160  CALL ibwrtf (brd0%,flname$)
   ```

   QuickBASIC/BASIC

   ```
   sta% = ilcmd (brd0%,"?@,",3)
   sta% = ilwrtf (brd0%,"Y.DAT")
   ```

   C

   ```
   ibcmd (brd0,"?@,",3);
   ibwrtf (brd0,"Y.DAT");
   ```

2. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

IBWRTI IBWRTI

Purpose: Write data from integer array.

Format:

BASICA

```
CALL ibwrti (ud%, iarr%(0), cnt%)
```

QuickBASIC/BASIC

```
CALL ibwrti (ud%, iarr%(), cnt%)
```

or

```
ilwrti (ud%, iarr%(), cnt%)
```

`ibwrti` is not available in C because it is not necessary. In C, the `ibwrt` call is adequate for sending data from any type of buffer. BASICA, QuickBASIC, and BASIC, which have more rigid typing rules, require a different call for array buffers than for strings.

`ud%` specifies a device or an interface board. `iarr%` is the integer array from which data is written. `cnt%` specifies the maximum number of bytes to be written. The data, stored as two-byte integers in `iarr%`, is sent in low-byte, high-byte order to the GPIB.

`ibwrti` is similar to the `ibwrt` function, which writes data from a character string variable. `ibwrti` can write up to 64 kilobytes-1 (2^{16}-1 bytes) of data.

Refer to *IBWRT* and to *BASICA/QuickBASIC/BASIC/C NI-488 I/O Calls and Functions* at the beginning of this chapter. Refer also to *IBWRTIA*.

IBWRTI (continued) IBWRTI

Device Function Example:

1. Write ten instruction bytes from the integer array `wrt` to `dvm`.

   BASICA

   ```
   100  DIM wrt%(4)
   110  wrt%(0) = ASC("F") + ASC("3") * 256
   120  wrt%(1) = ASC("R") + ASC("1") * 256
   130  wrt%(2) = ASC("X") + ASC("5") * 256
   140  wrt%(3) = ASC("P") + ASC("2") * 256
   150  wrt%(4) = ASC("G") + ASC("0") * 256
   160  cnt% = 10
   170  CALL ibwrti (dvm%,wrt%(0),cnt%)
   ```

   QuickBASIC/BASIC

   Replace line 170 with:

   ```
   CALL ibwrti (ud%, wrt%(), cnt%)
   ```

   or

   ```
   sta% = ilwrti (dvm%, wrt%(), 10)
   ```

2. Write five instruction bytes from integer array `wrt%` terminated by a carriage return and a linefeed to device `ptr%`. Linefeed is the EOS character of the device.

   BASICA

   ```
   100  DIM wrt%(3)
   110  wrt%(0) = ASC("I") + ASC("P") * 256
   120  wrt%(1) = ASC("2") + ASC("X") * 256
   130  wrt%(2) = ASC("5") + &H0D * 256
   140  wrt%(3) = &H0A
   150  cnt% = 7
   160  CALL ibwrti (ptr%, wrt%(0), cnt%)
   ```

IBWRTI (continued) IBWRTI

QuickBASIC/BASIC

Replace line 160 with:

```
CALL ibwrti (ptr%, wrt%(), cnt%)
```

or

```
DIM wrt% (3)
wrt% (0) = ASC ("I") + (ASC ("P") * 256)
wrt% (1) = ASC ("2") + (ASC ("X") * 256)
wrt% (2) = ASC ("5") + (&H0D * 256)
wrt% (3) = &H0A
sta% = ilwrti (ptr%, wrt%(), 7)
```

Board Function Examples:

1. Write ten instruction bytes from the integer array `wrt%` to a device at listen address hex 2F (ASCII `/`) (the GPIB board talk address is hex 40 or ASCII `@`).

   BASICA

```
100  REM  Perform addressing.
120  cmd$ = "?@/"     ' UNL MTA LAD
130  CALL ibcmd (brd0%,cmd$)
140  REM  Perform board write.
150  DIM wrt%(4)
160  wrt%(0) = ASC("F") + ASC("3") * 256
170  wrt%(1) = ASC("R") + ASC("1") * 256
180  wrt%(2) = ASC("X") + ASC("5") * 256
190  wrt%(3) = ASC("P") + ASC("2") * 256
200  wrt%(4) = ASC("G") + ASC("0") * 256
210  cnt% = 10
220  CALL ibwrti (brd0%,wrt%(0),cnt%)
```

   QuickBASIC/BASIC

   For QuickBASIC, replace line 220 with:

```
CALL ibwrti (brd0%, wrt%(), cnt%)
```

   or

```
sta% = ilwrti (brd0%, wrt%(), 10)
```

2. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

IBWRTIA IBWRTIA

Purpose: Write data asynchronously from integer array.

Format:

BASICA

```
CALL ibwrtia (ud%, iarr%(0), cnt%)
```

QuickBASIC/BASIC

```
CALL ibwrtia (ud%, iarr%(), cnt%)
```

or

```
ilwrtia (ud%, iarr%(), cnt%)
```

`ibwrtia` is not available in C because it is not necessary. In C, the `ibwrta` call is adequate for sending data from any type of buffer. BASICA, QuickBASIC, and BASIC, which have more rigid typing rules, require a different call for array buffers than for strings.

`ud%` specifies a device or an interface board. `iarr%` is the integer array from which data is written. `cnt%` specifies the maximum number of bytes to be written. The data is sent in low-byte, high-byte order.

`ibwrtia` is similar to the `ibwrta` function, which writes data from a character string variable. `ibwrtia` can write asynchronously up to 64 kilobytes-1 (2^{16}-1 bytes) of data.

Refer to *IBWRTA* and to *BASICA/QuickBASIC/BASIC/C NI-488 I/O Calls and Functions* at the beginning of this chapter.

Note: For QuickBASIC and BASIC, do not pass dynamic arrays to the asynchronous functions `ibrdia` and `ibwrtia` because their location may change during an I/O operation.

IBWRTIA (continued) IBWRTIA

Device Function Example:

1. Write ten instruction bytes from integer array to the device `dvm%` while performing other processing.

   BASICA

```
100  DIM wrt%(4)
110  wrt%(0) = ASC("F") + ASC("3") * 256
120  wrt%(1) = ASC("R") + ASC("1") * 256
130  wrt%(2) = ASC("X") + ASC("5") * 256
140  wrt%(3) = ASC("P") + ASC("2") * 256
150  wrt%(4) = ASC("G") + ASC("0") * 256
160  cnt% = 10
170  CALL ibwrtia (dvm%,wrt%(0),cnt%)
180  mask% = &H4100     ' TIMO CMPL
190  REM  Perform other processing here, then
200  REM  wait for I/O completion or timeout.
210  CALL ibwait (dvm%,mask%)
220  REM  ibsta% shows how the write terminated:
230  REM  CMPL, END, TIMO, or ERR.
250  REM  If CMPL is not set, continue
260  REM  processing.
270  IF (ibsta% AND &H100) = 0 GOTO 190
```

   QuickBASIC/BASIC

   Replace line 170 with:

```
CALL ibwrtia (dvm%, wrt%(), cnt%)
```

   or

```
sta% = ilwrtia (brd0%,wrt% (),10)
```

IBWRTIA (continued) IBWRTIA

Board Function Examples:

Write ten instruction bytes from the integer array `wrt` to a device at listen address hex 2F (ASCII `/`) and then unaddress (the GPIB board talk address is hex 40 or ASCII `@`).

BASICA

```
100  REM  Perform addressing.
110  cmd$ = "?@/" ' UNL MTA LAD
120  CALL ibcmd (brd0%,cmd$)
130  REM  Perform board write.
140  DIM wrt%(4)
150  wrt%(0) = ASC("F") + ASC("3") * 256
160  wrt%(1) = ASC("R") + ASC("1") * 256
170  wrt%(2) = ASC("X") + ASC("5") * 256
180  wrt%(3) = ASC("P") + ASC("2") * 256
190  wrt%(4) = ASC("G") + ASC("0") * 256
200  cnt% = 10
210  CALL ibwrtia (brd0%,wrt%(0),cnt%)
220  REM  Perform other processing here then
230  REM  wait for I/O completion or timeout.
240  mask% = &H4100     ' TIMO CMPL
250  CALL ibwait (brd0%,mask%)
```

QuickBASIC/BASIC

Replace line 210 with:

```
CALL ibwrtia (brd%, wrt%(), cnt%)
```

or

```
sta% = ilwrtia (brd0%,wrt%(),10)
```

2. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

BASICA/QuickBASIC/BASIC/C GPIB Programming Examples

These examples illustrate the programming steps that could be used to program a representative IEEE-488 instrument from your personal computer using the NI-488 functions. The applications are written in BASICA, QuickBASIC, BASIC, and C. The target instrument is a digital voltmeter (DVM). This instrument is otherwise unspecified (that is, it is not a DVM manufactured by any particular manufacturer). The purpose here is to explain how to use the driver to execute certain programming and control sequences and not how to determine those sequences.

Because the instructions that are sent to program a device as well as the data that might be returned from the device are called *device-dependent messages*, the format and syntax of the messages used in this example are unique to this device. Furthermore, the *interface messages* or bus commands that must be sent to each device will also vary, but to a lesser degree. The exact sequence of messages to program and to control a particular device are contained in its documentation.

For example, the following sequence of actions is assumed to be necessary to program this DVM to make and return measurements of a high frequency AC voltage signal in the autoranging mode:

1. Initialize the GPIB interface circuits of the DVM so that it can respond to messages.

2. Place the DVM in remote programming mode and turn off front panel control.

3. Initialize the internal measurement circuits.

4. Instruct the meter to measure volts alternating current (VAC) using auto-ranging (AUTO), to wait for a trigger from the Controller before starting a measurement (TRIGGER 2), and to assert the IEEE-488 Service Request signal line, SRQ, when the measurement has been completed and the meter is ready to send the result (*SRE 16).

5. For each measurement:

   a. Send the `TRIGGER` command to the multimeter. The `ibwrt` command `"VAL1?"` instructs the meter to send the next triggered reading to its IEEE-488 output buffer.

   b. Wait until the DVM asserts Service Request (SRQ) to indicate that the measurement is ready to be read.

   c. Serial poll the DVM to determine if the measured data is valid or if a fault condition exists. You can find out by checking the message available (MAV) bit, bit 4 in the status byte.

   d. If the data is valid, read 10 bytes from the DVM.

6. End the session.

The example programs that follow are based on these assumptions:

- The GPIB board is the designated System Active Controller of the GPIB.
- There is no change to the GPIB board default hardware settings.
- The only changes made to the software parameters are those necessary to define the device DVM at primary address 1.
- There is only one GPIB board in use, and it is designated GPIB0.
- The primary listen and talk addresses of GPIB0 are hex 20 (ASCII space character) and hex 40 (ASCII @ character), respectively.

BASICA Example Program–Device Functions

```
100  REM
110  REM You must merge this code with DECL.BAS.
120      CLS
130      print "READ MEASUREMENT FROM FLUKE 45..." : print
140  REM
150  REM Assign an unique identifier to the FLUKE 45 that you
160  REM configured using IBCONF.EXE.
170  REM
180      bdname$ = "DVM"
190      call ibfind(bdname$, dvm%)
200      msg$ = "IBFIND ERROR"
210      if (dvm% < 0) then GOSUB 6000 : STOP
220  REM
230  REM Clear the device.
240  REM
250      call ibclr (dvm%)
260      msg$ = "ibclr ERROR"
270      if (ibsta% and EERR) then GOSUB 6000 : STOP
280  REM
290  REM Write the function, range, and trigger source
300  REM instructions to the Fluke 45.
310  REM
320      wrt$ = "*RST; VAC; AUTO; TRIGGER 2; *SRE 16"
330      call ibwrt (dvm%,wrt$)
340      msg$ = "IBWRT ERROR"
350      if (ibsta% and EERR) then GOSUB 6000 : STOP
360  REM
370  REM Trigger the device and request measurement.
380  REM
390      call ibtrg (dvm%)
400      msg$ = "IBTRG ERROR"
410      if (ibsta% and EERR) then GOSUB 6000 : STOP
420  REM
430      wrt$ = "VAL1?"
440      call ibwrt(dvm%, wrt$)
450      msg$ = "ibwrt error"
460      if (ibsta% and EERR) then GOSUB 6000 : STOP
470  REM
480  REM Wait for the DVM to set RQS (hex 800) or for a timeout
490  REM (hex 4000).  These status bits are listed in Chapter 3
500  REM of the Software Reference Manual.  If the current time
510  REM limit is too short, use ibtmo to change it.
520  REM
530      print "Waiting for RQS..."
540      print
550      mask% = &H4800                   ' rqs + timo
560      call ibwait(dvm%, mask%)
570      msg$ = "ibwait error"
580      if (ibsta% and (EERR or TIMO)) then GOSUB 6000 : STOP
```

```
590  REM
600  REM If neither a timeout nor an error occurred, IBWAIT
must
610  REM have returned on RQS.  Now read the status byte.  If
it
620  REM is &H50, the Fluke 45 has valid data to send;
otherwise,
630  REM it has a fault condition to report.
640  REM
650      call ibrsp(dvm%,spr%)
660      msg$ = "IBRSP ERROR"
670      if (ibsta% and EERR) then GOSUB 6000 : STOP
680  REM
690      msg$ = "FLUKE 45 ERROR"
700      if spr% <> &H50 then GOSUB 7000 : STOP
710  REM
720  REM Read the measurement.
730  REM
740      rd$ = space$(10)
750      call ibrd (dvm%,rd$)
760      msg$ = "IBRD ERROR"
770      if (ibsta% and EERR) then GOSUB 6000 : STOP
780  REM
790      reading$ = left$(rd$, ibcnt%)
800      print "reading: "; reading$
810  REM
820  REM Call the ibonl function to disable the device DVM.
830  REM
840      v% = 0 : call ibonl (dvm%, v%) : STOP
850      END


6000 REM This routine would notify you that an IB call failed
6010 REM and print the status variables.
6020 REM
6030     Print msg$
6040 REM
6050     Print "ibsta= &H"; hex$(ibsta%); " <";
6060     If ibsta% and EERR  then print " ERR";
6070     If ibsta% and TIMO then print " TIMO";
6080     If ibsta% and EEND then print " END";
6090     If ibsta% and SRQI then print " SRQI";
6100     If ibsta% and RQS  then print " RQS";
6110     If ibsta% and CMPL then print " CMPL";
6120     If ibsta% and LOK  then print " LOK";
6130     If ibsta% and RREM then print " REM";
6140     If ibsta% and CIC  then print " CIC";
6150     If ibsta% and AATN then print " ATN";
6160     If ibsta% and TACS then print " TACS";
6170     If ibsta% and LACS then print " LACS";
6180     If ibsta% and DTAS then print " DTAS";
6190     If ibsta% and DCAS then print " DCAS";
6200     Print ">"
```

```
6210 REM
6220     Print "iberr= "; iberr%;
6230     If iberr% = EDVR then print " EDVR <DOS Error>"
6240     If iberr% = ECIC then print " ECIC <Not CIC>"
6250     If iberr% = ENOL then print " ENOL <No listener>"
6260     If iberr% = EADR then print " EADR <Address error>"
6270     If iberr% = EARG then print " EARG <Invalid argument>"
6280     If iberr% = ESAC then print " ESAC <Not Sys Ctrlr>"
6290     If iberr% = EABO then print " EABO <Op. aborted>"
6300     If iberr% = ENEB then print " ENEB <No GPIB board>"
6310     If iberr% = EOIP then print " EOIP <Async I/O in prg>"
6320     If iberr% = ECAP then print " ECAP <No capability>"
6330     If iberr% = EFSO then print " EFSO <File sys. error>"
6340     If iberr% = EBUS then print " EBUS <Command error>"
6350     If iberr% = ESTB then print " ESTB <Status byte lost>"
6360     If iberr% = ESRQ then print " ESRQ <SRQ stuck on>"
6370     If iberr% = ETAB then print " ETAB <Table Overflow>"
6380 REM
6390     Print "ibcnt = "; ibcnt%
6400 REM
6410 REM Call the ibonl function to disable the device DVM.
6420 REM
6430     v% = 0 : call ibonl (dvm%, v%) : RETURN


7000 REM This routine would notify you that the DVM returned an
7010 REM invalid serial poll response byte.
7020 REM
7030     print msg$
7040     print "Status Byte = "; spr%
7050 REM
7060 REM Call the ibonl function to disable the device DVM.
7070 REM
7080     v% = 0 : call ibonl (dvm%, v%) : RETURN
```

BASICA Example Program–Board Functions

```
150  REM
160  REM You must merge this code with DECL.BAS.
170      CLS
180      print "READ MEASUREMENT FROM FLUKE 45..." : print
190  REM
200  REM Assign a unique identifier to board 0 and store in
205  REM variable BRD0%.
210  REM
220      bdname$ = "GPIB0"
230      call ibfind(bdname$, brd0%)
240      msg$ = "IBFIND ERROR"
250      if brd0% < 0 then GOSUB 6000 : STOP
260  REM
270  REM Send the Interface Clear (IFC) message to all devices.
280  REM
290      call ibsic(brd0%)
300      msg$ = "IBSIC ERROR"
310      if (ibsta% and EERR) then GOSUB 6000 : STOP
320  REM
330  REM Turn on the Remote Enable (REN) signal.
340  REM
350      v% = 1 : call ibsre(brd0%, v%)
360      msg$ = "IBSRE ERROR"
370      if (ibsta% and EERR) then GOSUB 6000 : STOP
380  REM
390  REM Inhibit front panel control with the Local Lockout
400  REM (LLO) command (hex 11).  Place the Fluke 45 in remote
410  REM mode by addressing it to listen. The listen address of
420  REM device 1 (MLA1) is its primary address, 1, plus hex 20.
430  REM This is an ASCII "!".  Send the Device Clear (DCL)
440  REM message (hex 14) to clear internal device functions.
450  REM Finally, address the GPIB interface board to talk
(MTA0)
460  REM by sending its talk address, primary address 0 plus hex
465  REM 40, or ASCII "@".  These commands can be found in
465  REM Appendix A of the Software Reference Manual.
470  REM
480      cmd$ = CHR$(&H11) + "!" + CHR$(&H14) + "@"
490      call ibcmd(brd0%, cmd$)
500      msg$ = "IBCMD ERROR"
510      if (ibsta% and EERR) then GOSUB 6000 : STOP
520  REM
530  REM Write the function, range, and trigger source
540  REM instructions to the FLUKE 45.
550  REM
560      wrt$ = "*RST; VAC; AUTO; TRIGGER 2; *SRE 16"
570      call ibwrt(brd0%, wrt$)
580      msg$ = "IBWRT ERROR"
590      if (ibsta% and EERR) then GOSUB 6000 : STOP
```

```
600  REM
610  REM Send the GET message (hex 8) to trigger a
615  REM measurement reading.
620  REM
630      cmd$ = CHR$(&H8)                   ' GET
640      call ibcmd(brd0%, cmd$)
650      msg$ = "IBCMD ERROR"
660      if (ibsta% and EERR) then GOSUB 6000 : STOP
670  REM
680  REM Request triggered measurement reading.
690  REM
700      wrt$ = "VAL1?"
710      call ibwrt(brd0%, wrt$)
720      msg$ = "IBWRT ERROR"
730      if (ibsta% and EERR) then GOSUB 6000 : STOP
740  REM
750  REM Wait for the DVM to set SRQ (hex 1000) or for a timeout
760  REM (hex 4000).  These status bits are listed in Chapter 3
770  REM of the Software Reference Manual.  If the current time
780  REM limit is too short, use ibtmo to change it.
790  REM
800      print "Waiting for SRQ..."
810      print
820      mask% = &H5000                     ' SRQI + TIMO
830      call ibwait(brd0%, mask%)
840      msg$ = "IBWAIT ERROR"
850      if (ibsta% and (EERR and TIMO)) then GOSUB 6000 : STOP
860  REM
870  REM If neither a timeout nor an error occurred, IBWAIT must
880  REM have returned on SRQ.  Next, do a serial poll.  First
890  REM unaddress bus devices by sending the untalk (UNT)
900  REM command (ASCII "_", or hex 5F) and the unlisten (UNL)
910  REM command (ASCII "?", or hex 3F).  Then send the Serial
920  REM Poll Enable (SPE) command (hex 18) and the DVM's talk
930  REM address (MTA1) (device 1's primary address, 1, plus
940  REM hex 40, or ASCII "A") and the GPIB interface board
950  REM listen address (MLA0) (primary address 0 plus hex 20,
960  REM or ASCII space).  These commands can be found in
970  REM Appendix A of the Software Reference Manual.
980  REM
990      cmd$ = "_?" + CHR$(&H18) + "A "
1000     call ibcmd(brd0%, cmd$)
1010     msg$ = "IBCMD ERROR"
1020     if (ibsta% and EERR) then GOSUB 6000 : STOP
1030 REM
1040 REM Now read the status byte.  If it is &H50, the Fluke 45
1050 REM has valid data to send; otherwise, it has a fault
1060 REM condition to report.
1070 REM
1080     rd$ = space$(1)
1090     call ibrd(brd0%, rd$)
1100     msg$ = "ibrd error"
1110     if (ibsta% and EERR) then GOSUB 6000 : STOP
```

```
1120 REM
1130     msg$ = "Fluke 45 error"
1140     if asc(rd$) <> &H50 then GOSUB 7000 : STOP
1150 REM
1160 REM Complete the serial poll by sending the Serial Poll
1170 REM Disable (SPD) message, hex 19.  This command can be
1180 REM found in Appendix A of the Software Reference Manual.
1190 REM
1200     cmd$ = CHR$(&H19)                 ' SPD
1210     call ibcmd(brd0%, cmd$)
1220     msg$ = "IBCMD ERROR"
1230     if (ibsta% and EERR) then GOSUB 6000 : STOP
1240 REM
1250 REM Read the measurement.
1260 REM
1270     rd$ = space$(10)
1280     call ibrd(brd0%, rd$)
1290     msg$ = "IBRD ERROR"
1300     if (ibsta% and EERR) then GOSUB 6000 : STOP
1310 REM
1320     Reading$ = left$(Rd$, IBCNT%)
1330     Print "Reading: "; Reading$
1340 REM
1350 REM Call the ibonl function to disable the hardware and
1360 REM software.
1370 REM
1380     v% = 0 : call ibonl(brd0%, v%) : STOP
1390    END

6000 REM This routine would notify you that an ib call failed
6010 REM and print the status variables.
6020 REM
6030     Print msg$
6040 REM
6050     Print "ibsta= &H"; hex$(ibsta%); " <";
6060     If ibsta% and EERR  then print " ERR";
6070     If ibsta% and TIMO then print " TIMO";
6080     If ibsta% and EEND then print " END";
6090     If ibsta% and SRQI then print " SRQI";
6100     If ibsta% and RQS  then print " RQS";
6110     If ibsta% and CMPL then print " CMPL";
6120     If ibsta% and LOK  then print " LOK";
6130     If ibsta% and RREM then print " REM";
6140     If ibsta% and CIC  then print " CIC";
6150     If ibsta% and AATN then print " ATN";
6160     If ibsta% and TACS then print " TACS";
6170     If ibsta% and LACS then print " LACS";
6180     If ibsta% and DTAS then print " DTAS";
6190     If ibsta% and DCAS then print " DCAS";
6200     Print ">"
```

```
6210 REM
6220     Print "iberr= "; iberr%;
6230     If iberr% = EDVR then print " EDVR <DOS Error>"
6240     If iberr% = ECIC then print " ECIC <Not CIC>"
6250     If iberr% = ENOL then print " ENOL <No listener>"
6260     If iberr% = EADR then print " EADR <Address error>"
6270     If iberr% = EARG then print " EARG <Invalid argument>"
6280     If iberr% = ESAC then print " ESAC <Not Sys Ctrlr>"
6290     If iberr% = EABO then print " EABO <Op. aborted>"
6300     If iberr% = ENEB then print " ENEB <No GPIB board>"
6310     If iberr% = EOIP then print " EOIP <Async I/O in prg>"
6320     If iberr% = ECAP then print " ECAP <No capability>"
6330     If iberr% = EFSO then print " EFSO <File sys. error>"
6340     If iberr% = EBUS then print " EBUS <Command error>"
6350     If iberr% = ESTB then print " ESTB <Status byte lost>"
6360     If iberr% = ESRQ then print " ESRQ <SRQ stuck on>"
6370     If iberr% = ETAB then print " ETAB <Table Overflow>"
6380 REM
6390     Print "ibcnt = "; ibcnt%
6400 REM
6410 REM Call the ibonl function to disable the hardware and
6410 REM software.
6420 REM
6430     v% = 0 : call ibonl(brd0%, v%) : RETURN

7000 REM This routine would notify you that the device DVM
7010 REM returned an invalid serial poll response byte.
7020 REM
7030     print msg$
7040     print "Status Byte = ";rd$
7050 REM
7060 REM Call the ibonl function to disable the hardware and
7065 REM software.
7070 REM
7080     v% = 0 : call ibonl(brd0%, v%) : RETURN
```

QuickBASIC Example Program–Device Functions

```
   REM $include: 'qbdecl.bas'

   declare sub gpiberr(msg$)
   declare sub dvmerror(msg$)

   CLS
   PRINT "READ MEASUREMENT FROM FLUKE 45..."
   PRINT

'  Assign a unique identifier to the FLUKE 45 that you
'  configured using IBCONF.EXE.

   BDNAME$ = "DVM"
   CALL IBFIND(BDNAME$, DVM%)
   IF (DVM% < 0) THEN CALL GPIBERR("IBFIND ERROR")

'  Clear the device.

   CALL IBCLR (DVM%)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBCLR ERROR")

'  Write the function, range, and trigger source instructions
'  to the FLUKE 45.

   WRT$ = "*RST; VAC; AUTO; TRIGGER 2; *SRE 16"
   CALL IBWRT (DVM%,WRT$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBWRT ERROR")

'  Trigger the device and request measurement.

   CALL IBTRG (DVM%)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBTRG ERROR")

   WRT$ = "VAL1?"
   CALL IBWRT(DVM%, WRT$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBWRT ERROR")

'  Wait for the DVM to set RQS (hex 800) or for a timeout
'  (hex 4000).  These status bits are listed in Chapter 3 of
'  the Software Reference manual.  If the current time limit
'  is too short, use ibtmo to change it.

   print "Waiting for RQS..."
   print
   MASK% = &H4800                    ' RQS + TIMO
   CALL IBWAIT(DVM%, MASK%)
   IF (IBSTA% AND (EERR or TIMO)) THEN CALL GPIBERR("IBWAIT
   ERROR")
```

```
'  If neither a timeout nor an error occurred, IBWAIT must have
'  returned on RQS.  Now read the status byte.  If it is &H50,
'  the Fluke 45 has valid data to send; otherwise, it has a
'  fault condition to report.

   CALL IBRSP(DVM%,SPR%)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBRSP ERROR")

   IF SPR% <> &H50 THEN CALL DVMERR("FLUKE 45 ERROR")

'  Read the measurement.

   RD$ = SPACE$(10)
   CALL IBRD (DVM%,RD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBRD ERROR")

   Reading$ = left$(Rd$, IBCNT%)
   Print "Reading: "; Reading$

'  Call the ibonl function to disable the device DVM.

   CALL IBONL (DVM%,0)
END


'  This routine prints the result of the status variables.

SUB GPIBERR(MSG$) STATIC

   PRINT MSG$

   PRINT "IBSTA=&H"; HEX$(IBSTA%); " <";
   IF IBSTA% AND EERR THEN PRINT " ERR";
   IF IBSTA% AND TIMO THEN PRINT " TIMO";
   IF IBSTA% AND EEND THEN PRINT " END";
   IF IBSTA% AND SRQI THEN PRINT " SRQI";
   IF IBSTA% AND RQS  THEN PRINT " RQS";
   IF IBSTA% AND CMPL THEN PRINT " CMPL";
   IF IBSTA% AND LOK  THEN PRINT " LOK";
   IF IBSTA% AND RREM THEN PRINT " REM";
   IF IBSTA% AND CIC  THEN PRINT " CIC";
   IF IBSTA% AND AATN THEN PRINT " ATN";
   IF IBSTA% AND TACS THEN PRINT " TACS";
   IF IBSTA% AND LACS THEN PRINT " LACS";
   IF IBSTA% AND DTAS THEN PRINT " DTAS";
   IF IBSTA% AND DCAS THEN PRINT " DCAS";
   PRINT " >"
   PRINT "IBERR="; IBERR%;
   IF IBERR% = EDVR THEN PRINT " EDVR <DOS Error>"
   IF IBERR% = ECIC THEN PRINT " ECIC <Not CIC>"
   IF IBERR% = ENOL THEN PRINT " ENOL <No listener>"
   IF IBERR% = EADR THEN PRINT " EADR <Address error>"
   IF IBERR% = EARG THEN PRINT " EARG <Invalid argument>"
   IF IBERR% = ESAC THEN PRINT " ESAC <Not Sys Ctrlr>"
   IF IBERR% = EABO THEN PRINT " EABO <Op. aborted>"
   IF IBERR% = ENEB THEN PRINT " ENEB <No GPIB board>"
   IF IBERR% = EOIP THEN PRINT " EOIP <Async I/O in prg>"
   IF IBERR% = ECAP THEN PRINT " ECAP <No capability>"
   IF IBERR% = EFSO THEN PRINT " EFSO <File sys. error>"
   IF IBERR% = EBUS THEN PRINT " EBUS <Command error>"
   IF IBERR% = ESTB THEN PRINT " ESTB <Status byte lost>"
   IF IBERR% = ESRQ THEN PRINT " ESRQ <SRQ stuck on>"
   IF IBERR% = ETAB THEN PRINT " ETAB <Table Overflow>"

   PRINT "IBCNT="; IBCNT%

'  Call the ibonl function to disable the device DVM.

   CALL IBONL(dvm%, 0)
   STOP

END SUB


'  This routine would notify you that the DVM returned an
'  invalid serial poll response byte.

SUB DVMERR(MSG$) STATIC
   PRINT MSG$
   PRINT "Status Byte = ";SPR%

'  Call the ibonl function to disable the device DVM.

   CALL IBONL(dvm%, 0)
   STOP
END SUB
```

QuickBASIC Example Program–Board Functions

```
   REM $include: 'qbdecl.bas'

   declare sub gpiberr(msg$)
   declare sub dvmerr(msg$)

   CLS
   PRINT "READ MEASUREMENT FROM FLUKE 45..."
   PRINT

'  Assign an unique identifier to board 0 and store in variable
'  BRD0%.

   BDNAME$ = "GPIB0"
   CALL IBFIND(BDNAME$, BRD0%)
   IF BRD0% < 0 THEN CALL GPIBERR("IBFIND ERROR")

'  Send the Interface Clear (IFC) message to all devices.

   CALL IBSIC(BRD0%)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBSIC ERROR")

'  Turn on the Remote Enable (REN) signal.

   CALL IBSRE(BRD0%, 1)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBSRE ERROR")

'  Inhibit front panel control with the Local Lockout (LLO)
'  command (hex 11).  Place the Fluke 45 in remote mode by
'  addressing it to listen. The listen address of device 1
'  (MLA1) is its primary address, 1, plus hex 20.  This is
'  an ASCII "!".  Send the Device Clear (DCL) message (hex
'  14) to clear internal device functions.  Finally, address
'  the GPIB interface board to talk by sending its talk
'  address (MTA0), primary address  plus hex 40, or ASCII "@".
'  These commands can be found in Appendix A of the
'  Software Reference Manual.

   CMD$ = CHR$(&H11) + "!" + CHR$(&H14) + "@"
   CALL IBCMD(BRD0%, CMD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBCMD ERROR")

'  Write the function, range, and trigger source instructions to
'  the FLUKE 45.

   WRT$ = "*RST; VAC; AUTO; TRIGGER 2; *SRE 16"
   CALL IBWRT(BRD0%, WRT$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBWRT ERROR")
```

```
'  Send the GET message (hex 8) to trigger a measurement
reading.

   CMD$ = CHR$(&H8)                    ' GET
   CALL IBCMD(BRD0%, CMD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBCMD ERROR")

'  Request triggered measurement reading.

   WRT$ = "VAL1?"
   CALL IBWRT(BRD0%, WRT$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBWRT ERROR")

'  Wait for the DVM to set SRQ (hex 1000) or for a timeout
'  (hex 4000).  These status bits are listed in Chapter 3 of
'  the Software Reference manual.  If the current time limit
'  is too short, use IBTMO to change it.

   print "Waiting for SRQ..."
   print
   MASK% = &H5000                      ' SRQI + TIMO
   CALL IBWAIT(BRD0%, MASK%)
   IF (IBSTA% AND (EERR or TIMO) THEN
      CALL GPIBERR("IBWAIT ERROR")
   END IF

'  If neither a timeout nor an error occurred, IBWAIT must have
'  returned on SRQ.  Next do a serial poll.  First unaddress
'  bus devices by sending the untalk (UNT) command (ASCII "_",
'  or hex 5F) and the unlisten (UNL) command (ASCII "?", or hex
'  3F).  Then send the Serial Poll Enable (SPE) command (hex 18)
'  and the DVM's talk address (device 1's primary address, 1,
'  plus hex 40, or ASCII "A") and the GPIB interface board
'  listen address (primary address 0 plus hex 20, or ASCII
space).
'  These commands can be found in Appendix A of the Software
'  Reference Manual.

   CMD$ = "_?" + CHR$(&H18) + "A "    ' UNT, UNL, SPE, MTA1, MLA0
   CALL IBCMD(BRD0%, CMD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBCMD ERROR")

'  Now read the status byte.  If it is &H50, the Fluke 45 has
'  valid data to send; otherwise, it has a fault condition to
'  report.

   RD$ = SPACE$(1)
   CALL IBRD(BRD0%, RD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBRD ERROR")

   IF ASC(RD$) <> &H50 THEN
     CALL DVMERR("FLUKE 45 ERROR")
   END IF
```

```
'  Complete the serial poll by sending the Serial Poll Disable
'  (SPD) command, hex 19.  This command can be found in Appendix
'  A of the Software Reference Manual.

   CMD$ = CHR$(&H19)                 ' SPD
   CALL IBCMD(BRD0%, CMD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBCMD ERROR")

'  Read the measurement.

   RD$ = SPACE$(10)
   CALL IBRD(BRD0%, RD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBRD ERROR")

   Reading$ = left$(Rd$, IBCNT%)
   Print "Reading: "; Reading$

'  Call the ibonl function to disable the hardware and software.

   CALL IBONL(BRD0%, 0)
END


'  This routine prints the result of the status variables.

SUB GPIBERR(MSG$) STATIC

   PRINT MSG$

   PRINT "IBSTA=&H"; HEX$(IBSTA%); " <";
   IF IBSTA% AND EERR THEN PRINT " ERR";
   IF IBSTA% AND TIMO THEN PRINT " TIMO";
   IF IBSTA% AND EEND THEN PRINT " END";
   IF IBSTA% AND SRQI THEN PRINT " SRQI";
   IF IBSTA% AND RQS  THEN PRINT " RQS";
   IF IBSTA% AND CMPL THEN PRINT " CMPL";
   IF IBSTA% AND LOK  THEN PRINT " LOK";
   IF IBSTA% AND RREM THEN PRINT " REM";
   IF IBSTA% AND CIC  THEN PRINT " CIC";
   IF IBSTA% AND AATN THEN PRINT " ATN";
   IF IBSTA% AND TACS THEN PRINT " TACS";
   IF IBSTA% AND LACS THEN PRINT " LACS";
   IF IBSTA% AND DTAS THEN PRINT " DTAS";
   IF IBSTA% AND DCAS THEN PRINT " DCAS";
   PRINT " >"

   PRINT "IBERR="; IBERR%;
   IF IBERR% = EDVR THEN PRINT " EDVR <DOS Error>"
   IF IBERR% = ECIC THEN PRINT " ECIC <Not CIC>"
   IF IBERR% = ENOL THEN PRINT " ENOL <No listener>"
   IF IBERR% = EADR THEN PRINT " EADR <Address error>"
   IF IBERR% = EARG THEN PRINT " EARG <Invalid argument>"
   IF IBERR% = ESAC THEN PRINT " ESAC <Not Sys Ctrlr>"
   IF IBERR% = EABO THEN PRINT " EABO <Op. aborted>"
   IF IBERR% = ENEB THEN PRINT " ENEB <No GPIB board>"
   IF IBERR% = EOIP THEN PRINT " EOIP <Async I/O in prg>"
   IF IBERR% = ECAP THEN PRINT " ECAP <No capability>"
   IF IBERR% = EFSO THEN PRINT " EFSO <File sys. error>"
   IF IBERR% = EBUS THEN PRINT " EBUS <Command error>"
   IF IBERR% = ESTB THEN PRINT " ESTB <Status byte lost>"
   IF IBERR% = ESRQ THEN PRINT " ESRQ <SRQ stuck on>"
   IF IBERR% = ETAB THEN PRINT " ETAB <Table Overflow>"

   PRINT "IBCNT="; IBCNT%

'  Call the ibonl function to disable the hardware and software.

   CALL IBONL(BRD0%, 0)
   STOP

END SUB


'  This routine would notify you that the DVM returned an
'  invalid serial poll response byte.

SUB DVMERR(MSG$) STATIC
   PRINT MSG$
   PRINT "Status Byte = ";RD$

'  Call the ibonl function to disable the hardware and software.

   CALL IBONL(BRD0%, 0)
   STOP

END SUB
```

Microsoft BASIC Example Program–Device Functions

```
   REM $include: 'mbdecl.bas'

   declare sub gpiberr(msg$)
   declare sub dvmerr(msg$)

   CLS
   PRINT "READ MEASUREMENT FROM FLUKE 45..."
   PRINT

'  Assign an unique identifier to the FLUKE 45 that you
'  configured using IBCONF.EXE.

   BDNAME$ = "DVM"
   CALL IBFIND(BDNAME$, DVM%)
   IF (DVM% < 0) THEN CALL GPIBERR("IBFIND ERROR")

'  Clear the device.

   CALL IBCLR (DVM%)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBCLR ERROR")

'  Write the function, range, and trigger source instructions
'  to the FLUKE 45.

   WRT$ = "*RST; VAC; AUTO; TRIGGER 2; *SRE 16"
   CALL IBWRT (DVM%,WRT$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBWRT ERROR")

'  Trigger the device and request measurement.

   CALL IBTRG (DVM%)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBTRG ERROR")

   WRT$ = "VAL1?"
   CALL IBWRT(DVM%, WRT$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBWRT ERROR")

'  Wait for the DVM to set RQS (hex 800) or for a timeout (hex
'  4000).  These status bits are listed in Chapter 3 of the
'  Software Reference manual.  If the current time limit is too
'  short, use IBTMO to change it.

   PRINT "Waiting for RQS..."
   PRINT
   MASK% = &H4800                    ' RQS + TIMO
   CALL IBWAIT(DVM%, MASK%)
   IF (IBSTA% AND (EERR or TIMO)) THEN CALL GPIBERR("IBWAIT
   ERROR")
```

```
'  If neither a timeout nor an error occurred, IBWAIT must have
'  returned on RQS.  Now read the status byte.  If it is &H50,
'  the Fluke 45 has valid data to send; otherwise, it has a
'  fault condition to report.

   CALL IBRSP(DVM%,SPR%)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBRSP ERROR")

   IF SPR% <> &H50 THEN CALL DVMERR("FLUKE 45 ERROR")

'  Read the measurement.

   RD$ = SPACE$(10)
   CALL IBRD (DVM%,RD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBRD ERROR")

   READING$ = LEFT$(RD$, IBCNT%)
   PRINT "READING: "; READING$

'  Call the ibonl function to disable the device DVM.

   CALL IBONL (DVM%,0)

END


'  This routine prints the result of the status variables.

SUB GPIBERR(MSG$) STATIC

   PRINT MSG$

   PRINT "IBSTA=&H"; HEX$(IBSTA%); " <";
   IF IBSTA% AND EERR THEN PRINT " ERR";
   IF IBSTA% AND TIMO THEN PRINT " TIMO";
   IF IBSTA% AND EEND THEN PRINT " END";
   IF IBSTA% AND SRQI THEN PRINT " SRQI";
   IF IBSTA% AND RQS  THEN PRINT " RQS";
   IF IBSTA% AND CMPL THEN PRINT " CMPL";
   IF IBSTA% AND LOK  THEN PRINT " LOK";
   IF IBSTA% AND RREM THEN PRINT " REM";
   IF IBSTA% AND CIC  THEN PRINT " CIC";
   IF IBSTA% AND AATN THEN PRINT " ATN";
   IF IBSTA% AND TACS THEN PRINT " TACS";
   IF IBSTA% AND LACS THEN PRINT " LACS";
   IF IBSTA% AND DTAS THEN PRINT " DTAS";
   IF IBSTA% AND DCAS THEN PRINT " DCAS";
   PRINT " >"
   PRINT "IBERR="; IBERR%;
   IF IBERR% = EDVR THEN PRINT " EDVR <DOS Error>"
   IF IBERR% = ECIC THEN PRINT " ECIC <Not CIC>"
   IF IBERR% = ENOL THEN PRINT " ENOL <No listener>"
   IF IBERR% = EADR THEN PRINT " EADR <Address error>"
   IF IBERR% = EARG THEN PRINT " EARG <Invalid argument>"
   IF IBERR% = ESAC THEN PRINT " ESAC <Not Sys Ctrlr>"
   IF IBERR% = EABO THEN PRINT " EABO <Op. aborted>"
   IF IBERR% = ENEB THEN PRINT " ENEB <No GPIB board>"
   IF IBERR% = EOIP THEN PRINT " EOIP <Async I/O in prg>"
   IF IBERR% = ECAP THEN PRINT " ECAP <No capability>"
   IF IBERR% = EFSO THEN PRINT " EFSO <File sys. error>"
   IF IBERR% = EBUS THEN PRINT " EBUS <Command error>"
   IF IBERR% = ESTB THEN PRINT " ESTB <Status byte lost>"
   IF IBERR% = ESRQ THEN PRINT " ESRQ <SRQ stuck on>"
   IF IBERR% = ETAB THEN PRINT " ETAB <Table Overflow>"

   PRINT "IBCNT="; IBCNT%

'  Call the ibonl function to disable the device DVM.

   CALL IBONL (DVM%,0)
   STOP

END SUB


'  This routine would notify you that the DVM returned an
'  invalid serial poll response byte.

SUB DVMERR(MSG$) STATIC
   PRINT MSG$
   PRINT "Status byte = ";SPR%

'  Call the ibonl function to disable the device DVM.

   CALL IBONL (DVM%,0)
   STOP

END SUB
```

Microsoft BASIC Example Program–Board Functions

```
   REM $include: 'mbdecl.bas'

   declare sub gpiberr(msg$)
   declare sub dvmerr(msg$)

   CLS
   PRINT "READ MEASUREMENT FROM FLUKE 45..."
   PRINT

'  Assign a unique identifier to board 0 and store in variable
'  BRD0%.

   BDNAME$ = "GPIB0"
   CALL IBFIND(BDNAME$, BRD0%)
   IF BRD0% < 0 THEN CALL GPIBERR("IBFIND ERROR")

'  Send the Interface Clear (IFC) message to all devices.

   CALL IBSIC(BRD0%)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBSIC ERROR")

'  Turn on the Remote Enable (REN) signal.

   CALL IBSRE(BRD0%, 1)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBSRE ERROR")

'  Inhibit front panel control with the Local Lockout (LLO)
'  command (hex 11).  Place the Fluke 45 in remote mode by
'  addressing it to listen.  The listen address of device 1
'  (MLA1) is its primary address, 1, plus hex 20.  This is an
'  ASCII "!".  Send the Device Clear (DCL) message (hex 14) to
'  clear internal device functions.  Finally, address the GPIB
'  interface board to talk by sending its talk address (MTA0),
'  the primary address 0 plus hex 40, or ASCII "@".  These
'  commands can be found in Appendix A of the Software
'  Reference Manual.

   CMD$ = CHR$(&H11) + "!" + CHR$(&H14) + "@"
   CALL IBCMD(BRD0%, CMD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBCMD ERROR")

'  Write the function, range, and trigger source instructions
'  to the FLUKE 45.

   WRT$ = "*RST; VAC; AUTO; TRIGGER 2; *SRE 16"
   CALL IBWRT(BRD0%, WRT$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBWRT ERROR")

'  Send the GET message (hex 8) to trigger a measurement
'  reading.
```

```
   CMD$ = CHR$(&H8)                    ' GET
   CALL IBCMD(BRD0%, CMD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBCMD ERROR")

'  Request triggered measurement reading.

   WRT$ = "VAL1?"
   CALL IBWRT(BRD0%, WRT$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBWRT ERROR")

'  Wait for the DVM to set SRQ (hex 1000) or for a timeout
'  (hex 4000).  These status bits are listed in Chapter 3 of
'  the Software Reference Manual.  If the current time limit
'  is too short, use IBTMO to change it.

   PRINT "Waiting for SRQ..."
   PRINT
   MASK% = &H5000                      ' SRQI + TIMO
   CALL IBWAIT(BRD0%, MASK%)
   IF (IBSTA% AND (EERR or TIMO)) THEN
      CALL GPIBERR("IBWAIT ERROR")
   END IF

'  If neither a timeout nor an error occurred, IBWAIT must have
'  returned on SRQ.  Next, do a serial poll.  First unaddress
'  bus devices by sending the untalk (UNT) command (ASCII "_",
'  or hex 5F) and the unlisten (UNL) command (ASCII "?", or hex
'  3F).  Then send the Serial Poll Enable (SPE) command (hex 18)
'  and the DVM's talk address (device 1's primary address, 1,
'  plus hex 40, or ASCII "A") and the GPIB interface board
'  listen address (primary address 0 plus hex 20, or ASCII
space).
'  These commands can be found in Appendix A of the Software
'  Reference Manual.

    CMD$ = "_?" + CHR$(&H18) + "A "  ' UNT, UNL, SPE, MTA1, MLA0
    CALL IBCMD(BRD0%, CMD$)
    IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBCMD ERROR")

'  Now read the status byte.  If it is &H50, the Fluke 45 has
'  valid data to send; otherwise, it has a fault condition to
'  report.

   RD$ = SPACE$(1)
   CALL IBRD(BRD0%, RD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBRD ERROR")

   IF ASC(RD$) <> &H50 THEN
     CALL DVMERR("FLUKE 45 ERROR")
   END IF

'  Complete the serial poll by sending the Serial Poll Disable
'  (SPD) command, hex 19.  This command can be found in Appendix
'  A of the Software Reference Manual.
```

```
   CMD$ = CHR$(&H19)                 ' SPD
   CALL IBCMD(BRD0%, CMD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBCMD ERROR")

'  Read the measurement.

   RD$ = SPACE$(10)
   CALL IBRD(BRD0%, RD$)
   IF (IBSTA% AND EERR) THEN CALL GPIBERR("IBRD ERROR")

   READING$ = LEFT$(RD$, IBCNT%)
   PRINT "READING: "; READING$

'  Call the ibonl function to disable the hardware and software.

   CALL IBONL(BRD0%, 0)
END


'  This routine prints the result of the status variables.

SUB GPIBERR(MSG$) STATIC

   PRINT MSG$

   PRINT "IBSTA=&H"; HEX$(IBSTA%); " <";
   IF IBSTA% AND EERR THEN PRINT " ERR";
   IF IBSTA% AND TIMO THEN PRINT " TIMO";
   IF IBSTA% AND EEND THEN PRINT " END";
   IF IBSTA% AND SRQI THEN PRINT " SRQI";
   IF IBSTA% AND RQS  THEN PRINT " RQS";
   IF IBSTA% AND CMPL THEN PRINT " CMPL";
   IF IBSTA% AND LOK  THEN PRINT " LOK";
   IF IBSTA% AND RREM THEN PRINT " REM";
   IF IBSTA% AND CIC  THEN PRINT " CIC";
   IF IBSTA% AND AATN THEN PRINT " ATN";
   IF IBSTA% AND TACS THEN PRINT " TACS";
   IF IBSTA% AND LACS THEN PRINT " LACS";
   IF IBSTA% AND DTAS THEN PRINT " DTAS";
   IF IBSTA% AND DCAS THEN PRINT " DCAS";
   PRINT " >"

   PRINT "IBERR="; IBERR%;
   IF IBERR% = EDVR THEN PRINT " EDVR <DOS Error>"
   IF IBERR% = ECIC THEN PRINT " ECIC <Not CIC>"
   IF IBERR% = ENOL THEN PRINT " ENOL <No listener>"
   IF IBERR% = EADR THEN PRINT " EADR <Address error>"
   IF IBERR% = EARG THEN PRINT " EARG <Invalid argument>"
   IF IBERR% = ESAC THEN PRINT " ESAC <Not Sys Ctrlr>"
   IF IBERR% = EABO THEN PRINT " EABO <Op. aborted>"
   IF IBERR% = ENEB THEN PRINT " ENEB <No GPIB board>"
   IF IBERR% = EOIP THEN PRINT " EOIP <Async I/O in prg>"
   IF IBERR% = ECAP THEN PRINT " ECAP <No capability>"
   IF IBERR% = EFSO THEN PRINT " EFSO <File sys. error>"
   IF IBERR% = EBUS THEN PRINT " EBUS <Command error>"
   IF IBERR% = ESTB THEN PRINT " ESTB <Status byte lost>"
   IF IBERR% = ESRQ THEN PRINT " ESRQ <SRQ stuck on>"
   IF IBERR% = ETAB THEN PRINT " ETAB <Table Overflow>"

   PRINT "IBCNT="; IBCNT%

'  Call the ibonl function to disable the hardware and software.

   CALL IBONL(BRD0%, 0)
   STOP
END SUB


'  This routine would notify you that the DVM returned an
'  invalid serial poll response byte.

SUB DVMERR(MSG$) STATIC
   PRINT MSG$
   PRINT "Status Byte = ";RD$

'  Call the ibonl function to disable the hardware and software.

   CALL IBONL(BRD0%, 0)
   STOP
END SUB
```

C Example Program – Device Functions

```
#include <stdio.h>
#include <stdlib.h>
#include "decl.h"

void dvmerr(char *msg, char code);  /* device error function */
void gpiberr(char *msg);            /* gpib error function   */


/* Application program variables passed to GPIB functions    */

   char rd[512];                /* read data buffer          */
   int  dvm;                    /* device number             */
   char spr;                    /* serial poll response byte */


main() {
    system("cls");

    printf("READ MEASUREMENT FROM FLUKE 45...\n");
    printf("\n");

 /* Assign a unique identifier to the Fluke 45 that you
    configured using ibconf.exe.                             */

    if ((dvm = ibfind ("DVM")) < 0) {
       gpiberr("ibfind Error");
       exit(1);
       }

 /* Clear the device.                                        */

    if (ibclr (dvm) & ERR) {
       gpiberr("ibclr Error");
       exit(1);
       }

 /* Write the function, range, and trigger source
    instructions to the DVM.                                 */

    ibwrt (dvm,"*RST; VAC; AUTO; TRIGGER 2; *SRE 16", 35L);
    if (ibsta & ERR) {
       gpiberr("ibwrt Error");
       exit(1);
       }

 /* Trigger the device and request measurement.              */

    if (ibtrg (dvm) & ERR) {
       gpiberr("ibtrg Error");
       exit(1);
       }
    ibwrt (dvm,"VAL1?", 5L);
    if (ibsta & ERR) {
       gpiberr("ibwrt Error");
       exit(1);
       }

 /* Wait for the DVM to set RQS or for a timeout; if the
    current time limit is too short, use ibtmo to change it. */

    printf("Waiting for RQS...\n");
    printf("\n");
    if (ibwait (dvm,TIMO|RQS) & (ERR|TIMO)) {
       gpiberr("ibwait Error");
       exit(1);
       }

 /* Because neither a timeout nor an error occurred, ibwait
    must have returned on RQS.  Next, serial poll the device.*/

    if (ibrsp (dvm, &spr) & ERR) {
       gpiberr("ibrsp Error");
       exit(1);
       }

 /* Now test the status byte.  If spr is 0x50, the Fluke 45
    has valid data to send; otherwise, it has a fault
    condition to report.                                    */

    if (spr != 0x50) {
       dvmerr("Fluke 45 Error", spr);
       exit(1);
       }

 /* If the data is valid, read the measurement.             */

    if (ibrd (dvm,rd,10L) & ERR) {
       gpiberr("ibrd Error");
       exit(1);
       }

    rd[ibcnt] = '\0';

    printf("Reading :  %s\n", rd);

 /* Call the ibonl function to disable device DVM.          */

    ibonl (dvm,0);
    }

void gpiberr(char *msg) {
```

```
/* This routine would notify you that an ib call failed.    */

   printf ("%s\n", msg);

   printf ( "ibsta=&H%x  <", ibsta);
   if (ibsta & ERR ) printf (" ERR");
   if (ibsta & TIMO) printf (" TIMO");
   if (ibsta & END ) printf (" END");
   if (ibsta & SRQI) printf (" SRQI");
   if (ibsta & RQS ) printf (" RQS");
   if (ibsta & CMPL) printf (" CMPL");
   if (ibsta & LOK ) printf (" LOK");
   if (ibsta & REM ) printf (" REM");
   if (ibsta & CIC ) printf (" CIC");
   if (ibsta & ATN ) printf (" ATN");
   if (ibsta & TACS) printf (" TACS");
   if (ibsta & LACS) printf (" LACS");
   if (ibsta & DTAS) printf (" DTAS");
   if (ibsta & DCAS) printf (" DCAS");
   printf (" >\n");

   printf ("iberr= %d", iberr);
   if (iberr == EDVR) printf (" EDVR <DOS Error>\n");
   if (iberr == ECIC) printf (" ECIC <Not CIC>\n");
   if (iberr == ENOL) printf (" ENOL <No listener>\n");
   if (iberr == EADR) printf (" EADR <Address error>\n");
   if (iberr == EARG) printf (" EARG <Invalid argument>\n");
   if (iberr == ESAC) printf (" ESAC <Not Sys Ctrlr>\n");
   if (iberr == EABO) printf (" EABO <Op. aborted>\n");

   if (iberr == ENEB) printf (" ENEB <No GPIB board>\n");
   if (iberr == EOIP) printf (" EOIP <Async I/O in prg>\n");
   if (iberr == ECAP) printf (" ECAP <No capability>\n");
   if (iberr == EFSO) printf (" EFSO <File sys. error>\n");
   if (iberr == EBUS) printf (" EBUS <Command error>\n");
   if (iberr == ESTB) printf (" ESTB <Status byte lost>\n");
   if (iberr == ESRQ) printf (" ESRQ <SRQ stuck on>\n");
   if (iberr == ETAB) printf (" ETAB <Table Overflow>\n");

   printf ("ibcnt= %d\n", ibcntl);
   printf ("\n");

 /* Call the ibonl function to disable device DVM.          */

    ibonl (dvm,0);
    }


void dvmerr(char *msg,  char spr) {
```

```
/* This routine would notify you that the DVM returned an
   invalid serial poll response byte.                        */

   printf ("%s\n", msg);
   printf("Status Byte = %x\n", spr);

 /* Call the ibonl function to disable device DVM.           */

   ibonl (dvm,0);
   }
```

C Example Program – Board Functions

```
#include <stdio.h>
#include "decl.h"

/* Application program variables passed to GPIB functions    */

  char rd[512];                    /* read data buffer         */
  int  bd;                         /* board or device number   */

void dvmerr(char *msg, char *code);  /* device error function
                                                               */
void gpiberr(char *msg);             /* gpib error function  */


main() {
   system("cls");

   printf("READ MEASUREMENT FROM FLUKE 45...\n");
   printf("\n");

/* Assign an unique identifier to board 0 and store in
/* variable bd.                                                */

   if ((bd = ibfind ("GPIB0")) < 0) {
      gpiberr("ibfind Error");
      exit(1);
      }

/* Send the Interface Clear (IFC) message to all devices.    */

   if (ibsic (bd) & ERR) {
      gpiberr("ibsic Error");
      exit(1);
      }

/* Turn on the Remote Enable (REN) signal.                     */

   if (ibsre (bd,1) & ERR) {
      gpiberr("ibsre Error");
      exit(1);
      }

/* Inhibit front panel control with the Local Lockout (LLO)
   command, place the Fluke 45 in remote mode by addressing it
   to listen, send the Device Clear (DCL) message to clear
   internal device functions, and address the GPIB board to
   talk.                                                       */

   ibcmd (bd,"\021!\024@",4L);
   if (ibsta & ERR) {
      gpiberr("ibcmd Error");
      exit(1);
      }
```

```
/* Write the function, range, and trigger source
   instructions to Fluke 45.                               */

   ibwrt (bd,"*RST; VAC; AUTO; TRIGGER 2; *SRE 16", 35L);
   if (ibsta & ERR) {
      gpiberr("ibwrt Error");
      exit(1);
      }

/* Send the GET message to trigger a measurement reading.   */

   ibcmd (bd,"\010",1L);
   if (ibsta & ERR) {
      gpiberr("ibcmd Error");
      exit(1);
      }

/* Request the triggered measurement reading.               */

   ibwrt (bd,"VAL1?", 5L);
   if (ibsta & ERR) {
      gpiberr("ibwrt Error");
      exit(1);
      }

/* Wait for the DVM to set SRQ or for a timeout; if the current
   time limit is too short, use ibtmo to change it.         */

   printf("Waiting for SRQ...\n");
   printf("\n");
   if (ibwait (bd,TIMO|SRQI) & (ERR|TIMO)) {
      gpiberr("ibwait Error");
      exit(1);
      }

/* Because neither a timeout nor an error occurred, ibwait must
   have returned on SRQ.  Next, do a serial poll.  First,
   unaddress bus devices and send the Serial Poll Enable (SPE)
   command, followed the talk address of the Fluke 45, and the
   listen address of the GPIB board.                        */

   ibcmd (bd,"?_\030A ",5L);     /* UNL UNT SPE TAD MLA      */
   if (ibsta & ERR) {
      gpiberr("ibcmd Error");
      exit(1);
      }

/* Now read the status byte.  If it is 0x50, the Fluke 45 has
   valid data to send; otherwise, it has a fault condition to
   report.                                                  */
   if (ibrd (bd,rd,1L) & ERR) {
      gpiberr("ibrd Error");
      exit(1);
      }

   if (rd[0] != 0x50) {
      dvmerr("Fluke 45 Error", rd);
      exit(1);
      }

/* Complete the serial poll by sending the Serial Poll Disable
   (SPD) message.                                              */

   if (ibcmd (bd,"\031",1L) & ERR) {
      gpiberr("ibcmd Error");
      exit(1);
      }

/* Because the DVM and GPIB board are still addressed to talk
   and listen, the measurement can be read as follows:         */

   if (ibrd (bd,rd,10L) & ERR) {
      gpiberr("ibrd Error");
      exit(1);
      }

   rd[ibcnt] = '\0';

   printf("Reading :  %s\n", rd);

/* Call the ibonl function to disable the hardware and
   software.                                                   */

   ibonl (bd,0);
   }


void gpiberr(char *msg) {

/* This routine would notify you that an ib call failed.       */

   printf ("%s\n", msg);

   printf ( "ibsta=&H%x  <", ibsta);
   if (ibsta & ERR ) printf (" ERR");
   if (ibsta & TIMO) printf (" TIMO");
   if (ibsta & END ) printf (" END");
   if (ibsta & SRQI) printf (" SRQI");
   if (ibsta & RQS ) printf (" RQS");
   if (ibsta & CMPL) printf (" CMPL");
   if (ibsta & LOK ) printf (" LOK");
   if (ibsta & REM ) printf (" REM");
   if (ibsta & CIC ) printf (" CIC");
   if (ibsta & ATN ) printf (" ATN");
   if (ibsta & TACS) printf (" TACS");
   if (ibsta & LACS) printf (" LACS");
   if (ibsta & DTAS) printf (" DTAS");
   if (ibsta & DCAS) printf (" DCAS");
   printf (" >\n");

   printf ("iberr= %d", iberr);
   if (iberr == EDVR) printf (" EDVR <DOS Error>\n");
   if (iberr == ECIC) printf (" ECIC <Not CIC>\n");
   if (iberr == ENOL) printf (" ENOL <No listener>\n");
   if (iberr == EADR) printf (" EADR <Address error>\n");
   if (iberr == EARG) printf (" EARG <Invalid argument>\n");
   if (iberr == ESAC) printf (" ESAC <Not Sys Ctrlr>\n");
   if (iberr == EABO) printf (" EABO <Op. aborted>\n");
   if (iberr == ENEB) printf (" ENEB <No GPIB board>\n");
   if (iberr == EOIP) printf (" EOIP <Async I/O in prg>\n");
   if (iberr == ECAP) printf (" ECAP <No capability>\n");
   if (iberr == EFSO) printf (" EFSO <File sys. error>\n");
   if (iberr == EBUS) printf (" EBUS <Command error>\n");
   if (iberr == ESTB) printf (" ESTB <Status byte lost>\n");
   if (iberr == ESRQ) printf (" ESRQ <SRQ stuck on>\n");
   if (iberr == ETAB) printf (" ETAB <Table Overflow>\n");

   printf ("ibcnt= %d\n", ibcntl);
   printf ("\n");

/* Call the ibonl function to disable the hardware
   and software.                                              */

   ibonl (bd,0);
   }


void dvmerr(char *msg,char *code) {

/* This routine would notify you that the DVM returned an
   invalid serial poll response byte.                         */

   printf ("%s\n", msg);
   printf("Status byte = %x\n", code[0]);

/* Call the ibonl function to disable the hardware
   and software.                                              */

   ibonl (bd,0);
   }
```

Chapter 6
IBIC

This chapter contains instructions for running the Interface Bus Interactive Control (IBIC) program using the NI-488.2 routines and NI-488 functions. It also contains the syntax of these functions, important characteristics, and IBIC sample programs.

With the Interface Bus Interactive Control (IBIC) program, you can communicate with the GPIB devices through functions you enter at the keyboard. This feature helps you learn how to communicate with the device, troubleshoot problems, and develop your application.

IBIC functions include most of the NI-488 functions and NI-488.2 routines described in Chapters 4 and 5, respectively, plus auxiliary functions used only by IBIC. IL functions (QuickBasic Function Format) are not supported in IBIC.

In IBIC, the user can send data and GPIB commands to a device from the keyboard and display data received from a device on the screen. After each command executes, the numeric value and mnemonic representation of the status word ibsta is displayed. The byte count ibcnt and error code iberr are also shown when appropriate.

This interactive method of data input and data/status output is designed to help you learn how to use the NI-488 functions and NI-488.2 routines to program your device. Once you develop a sequence of steps that works successfully for your system, you can easily incorporate the sequence into an application program using the appropriate language and syntax described in Chapters 3, 4, and 5.

Running IBIC

The `IBIC` program, `IBIC.EXE`, is an executable file that was copied from the distribution disk to the appropriate subdirectory (depending on which interface board you have) when you ran `INSTALL`. This example uses `GPIB-PC` as the appropriate subdirectory.

To run `IBIC`, change the directory to this subdirectory and enter *`ibic`* at the prompt.

Note: User inputs must be followed by pressing <Enter>. User inputs are italicized in all the examples in this section.

```
C:\GPIB-PC>     ibic



National Instruments
IEEE-488 Interface Bus Interactive Control Program (IBIC)
Copyright © 1990 National Instruments, Inc. , Version 1.0
All Rights Reserved

Type "help" for help.
```

Messages display on the screen that give you information about the `HELP`, `ibfind`, and `SET` commands.

The first input prompt to `IBIC` is a colon (`:`).

Using NI-488.2 Routines

The `SET` function is used to select the NI-488.2 function mode. The syntax for this form of the `SET` command is:

```
        set 488.2  n
```

where `n` represents a board number (for example, `n`=1 for `gpib1`).

Note: The default value of `n` is 0 (`gpib0`). Other board indexes can be used here also.

After issuing this form of the `SET` command, `IBIC` uses the `488.2` prompt to remind you that you are in NI-488.2 mode on board `n`.

```
set 488.2 1
488.2 (1):
```

After issuing the `set 488.2` command, any of the 488.2 routines can be used. The syntax of the NI-488.2 routines is shown in Table 6-2.

Using Send

The `Send` routine sends data to a single GPIB device. The `SendList` command can be used to send data to multiple GPIB devices. For example, to send the five character string `*IDN?` followed by the NL character with EOI from the computer to the devices at primary address 2 and 17, enter the following command at the `488.2 (0):` prompt.

```
488.2 (0):   SendList 2, 17 "*IDN?" NLend
[0128] (cmpl cic tacs)
count: 6
```

The returned status word `[0128]` indicates a successful I/O completion, while the byte count indicates that all six characters were sent from the computer and received by both devices.

Using Receive

The `Receive` routine causes the GPIB board to receive data from another GPIB device. The following example illustrates the use of the `Receive` routine.

```
488.2 (0):   Receive 5 10 STOPend
[2124] (end cmpl cic lacs)
count: 5
 48 65 6c 6c 6f          H e l l o
```

The command acquires data from the device at primary address 5. It stops receiving data when ten (10) characters have been received or when the END message is received. The acquired data is then displayed in hex format along with its ASCII equivalent. The status word and byte count are also displayed.

Using NI-488 Functions

In using `IBIC`, the most important NI-488 functions are the `HELP`, `ibfind` or `ibdev`, `ibwrt`, and `ibrd` commands. These functions are described in the following paragraphs.

Using HELP

The `HELP` function gives online information about `IBIC` and the functions available within the environment. This facility provides a quick reference for checking the syntax and function of the GPIB call.

Using IBFIND

To execute any NI-488 GPIB function, you must first use `ibfind` to open the device or board you want to use. When the device or board is opened, the symbolic name of that device or board is added to the prompt. If you are using NI-488.2 routines, you do not need to call `ibfind`: you may go directly to NI-488.2 mode with the `SET` command (see below).

The following examples show `ibfind` opening `dev1` (Example 1) and `gpib0` (Example 2).

Example 1:

```
: ibfind dev1
id = 32005

dev1:
```

Example 2:

```
: ibfind gpib0
id = 32005

gpib0:
```

The name used with the `ibfind` function must be a valid symbolic name known by the driver, as described in `IBCONF` (refer to *IBCONF* in Chapter 2). Both `dev1` and `gpib0` are default names found in the driver. `IBIC` makes no distinction between uppercase and lowercase.

Using IBDEV

To execute any GPIB function, you must first use `ibfind` or `ibdev` to open and initialize an unused device. The `ibdev` command selects an unopened device, opens it, and initializes its access board and the following fields to the values that are input:

- Primary Address
- Secondary Address
- Timeout Setting
- EOT
- EOS

Example 1 shows `ibdev` opening an available device and assigning it to access `gpib0` (`board` = 0) with a primary address of 6 (`pad` = 6), a secondary address of hex 67 (`sad` = 0x67), a timeout of 10 msec. (`tmo`=7), the END message enabled (`eot` =1), and the EOS mode disabled (`eos`= 0).

Example 1:

```
: ibdev 0 6 0x67 7 1 0
id = 32006

ud0:
```

If you simply type

`ibdev` <Enter>

you will be prompted for the input parameters, as shown in Example 2.

Example 2:

```
: ibdev
    enter board index:   0
    enter primary address:    6
    enter secondary address:    0x67
    enter timeout:   7
    enter 'EOI on last byte' flag:    1
    enter end-of-string mode/byte:    0
id = 32006

ud0:
```

There are three distinct errors that can occur with the `ibdev` call:

- If no device is available, an EDVR error is returned. This is also the error that occurs if the specified board index refers to a nonexistent board (that is, not 0 or 1). Example 3 illustrates an EDVR error.

  Example 3:

  ```
  : ibdev 4 6 0x67 7 1 0
  [8000]  ( err )
  error: EDVR (-1)

  :
  ```

- If the specified board index refers to a known board (such as 0 or 1) but the board cannot be found by the driver, an ENEB error is returned. In this case, run `IBCONF` to ensure that the base address of the board is set correctly.

- If one of the last five parameters is an illegal value, the `ibdev` call returns with a new `udx` prompt and the EARG error (invalid function argument). Example 4 illustrates an EARG error.

  Example 4:

  ```
  : ibdev 0 66 0x67 7 1 0
  [8100]  ( err cmpl )
  error: EARG

  ud0:  ibpad 6
  previous value: 16
  ```

  If the `ibdev` call returns with an EARG error, it is necessary to identify which parameters are incorrect and use the appropriate command to fix it. In *Example 4*, the pad is wrong and can be fixed with an `ibpad` call.

Note: The `ibdev` call should not be used until after all `ibfind` calls have been made for devices. This is to ensure that `ibdev` does not choose a device that may later be used with an `ibfind` call. In addition, all device descriptors are valid until they are explicitly taken offline by an `ibonl 0` call. Therefore, it is important to place all devices offline at the end of an application in order to ensure that `ibdev` functions correctly in subsequent applications.

Using IBWRT

The `ibwrt` command sends data to a GPIB device. For example, to send the six character data string `F3R5T1` from the computer to a device called `dev1`, you enter the following string at the `dev1:` prompt.

Example:

```
dev1: ibwrt "F3R5T1"
[0100] (cmpl)
count:  6
```

The returned Status Word [0100] indicates a successful I/O completion, while the Byte Count indicates that all six characters were sent from the computer and received by the device.

Using IBRD

The `ibrd` command causes a GPIB device to receive data from another GPIB device. The following example illustrates the use of the `ibrd` function.

Example:

```
dev1:    ibrd 20
[2100] (end cmpl)
count:  18
4E 44 43 56 28 30 30 30          N D C V ( 0 0 0
2E 30 30 34 37 45 2B 30          . 0 0 4 7 E + 0
0D 0A                            • •
```

This command acquires data from the device and displays it on the screen in hex format and in its ASCII equivalent, along with information about the data transfer such as the Status Word and the Byte Count.

How to Exit IBIC

Typing `e` or `q` returns you to the DOS operating system.

Adding Common EOS Characters

Some GPIB instruments require special termination characters or End-of-String (EOS) characters to indicate to the device the end of transmission. If your device requires any EOS characters, you must add these to the end of the data string sent out by the `ibwrt` statement.

The following example illustrates the addition of the two most commonly used EOS characters, the carriage return and the linefeed.

Example:

```
dev1:   ibwrt "F3R5T1\r\n"
[0100] (cmpl)
count: 6
```

The `\r` and `\n` represent the carriage return and linefeed characters respectively. See the Notes for Table 6-4 for a more detailed description on the representation of non-printable characters.

Using SET

If you are using NI-488 calls, you use `ibfind` to open each device or board. Once the device or board is opened, use the auxiliary function `SET` to select which opened device or board to access. `SET` changes the prompt to the new symbolic name. `SET` is also used to switch between NI-488 mode and NI-488.2 mode.

Example:

```
dev1: set plotter
id: 32006

plotter:
```

This example assumes that `IBCONF` was used to give a device the name `plotter`.

The following example summarizes the use of `ibfind` and `SET` in a typical program.

Example:

```
: ibfind dev1
id = 32006
dev1:   ibfind plotter

plotter:    ibwrt "F3T7G0"
[0100]   (cmpl)
count: 6

plotter:   set dev1

dev1:   ibwrt "X7Y39G0"
[0100]   (cmpl)
count: 7

dev1:
```

IBIC Functions and Syntax

`IBIC` displays the following information about each function call immediately after that call:

- `ibrd` and `ibrda` data messages are displayed on the screen in hex and ASCII formats.
- The global variables `ibsta`, `ibcnt`, and `iberr` are displayed on the screen.

`IBIC` and programming languages of Chapter 5 differ in the syntax of the function call. The main differences are that `ibwrt`, `ibwrta`, `ibcmd`, and `ibcmda` messages are entered as strings from the keyboard. The syntax for `IBIC` is shown in Tables 6-1 and 6-2.

The unit descriptor (`ud`) is not explicitly a part of `IBIC` function syntax. Before using any device or board, first call `ibfind` to open that unit and to pass the unit descriptor to `IBIC`. The screen prompt identifies which of these opened units `IBIC` will use in subsequent calls. Use the `SET` function to change from one of these units to another.

Tables 6-1 and 6-2 summarize the NI-488 functions and syntax and NI-488.2 routines, respectively, when called from `IBIC`. Syntax rules for `IBIC` are explained in the table notes. Consult Chapters 4 and 5 for detailed descriptions of functions and routines, respectively, and for syntax rules of the programming language you will use.

Table 6-1. Syntax of GPIB Functions in IBIC

Function Syntax	**Description**	**Function Type**	**Note**
`ibbna brdname`	Change access board of device	dev	1
`ibcac [v]`	Become active Controller	brd	2,3
`ibclr`	Clear specified device	dev	
`ibcmd string`	Send commands from string	brd	4
`ibcmda string`	Send commands asynch. from string	brd	4
`ibconfig mn v`	Alter configurable parameters	dev,brd	15,3
`ibdev v v v v v v`	Open an unused device when the device name is unknown	dev	9
`ibdma [v]`	Enable/disable DMA	brd	2,3
`ibeos v`	Change/disable EOS message	dev, brd	2,3
`ibeot [v]`	Enable/disable END message	dev, brd	2,3
`ibevent`	Return the next event	brd	
`ibfind udname`	Return unit descriptor	dev, brd	5
`ibgts [v]`	Go from active Controller to standby	brd	2,3
`ibist [v]`	Set/clear ist	brd	2,3
`iblines`	Read the state of all GPIB lines	dev, brd	

(continues)

Table 6-1. Syntax of GPIB Functions in IBIC (continued)

Function Syntax	Description	Function Type	Note
`ibln v v`	Check for presence of device on bus	dev, brd	10
`ibloc`	Go to local	dev, brd	
`ibonl [v]`	Place device online or offline	dev, brd	2,3
`ibpad v`	Change primary address	dev, brd	3
`ibpct`	Pass control	dev	
`ibppc v`	Parallel poll configure	dev, brd	3
`ibrd v`	Read data	dev, brd	6
`ibrda v`	Read data asynchronously	dev, brd	6
`ibrdf flname`	Read data to file	dev, brd	7
`ibrpp`	Conduct a parallel poll	dev, brd	
`ibrsc [v]`	Request/release system control	brd	2,3
`ibrsp`	Return serial poll byte	dev	
`ibrsv v`	Request service	dev	3
`ibsad v`	Change secondary address	dev, brd	3
`ibsic`	Send interface clear	brd	3
`ibsre [v]`	Set/clear remote enable line	brd	2,3
`ibstop`	Abort asynchronous operation	dev, brd	
`ibtmo v`	Change/disable time limit	dev, brd	3
`ibtrap mask v`	Configure applications monitor	dev, brd	3,8
`ibtrg`	Trigger selected device	dev	
`ibwait [mask]`	Wait for selected event	dev, brd	2,8
`ibwrt string`	Write data	brd	4
`ibwrta string`	Write data asynchronously	dev, brd	4
`ibwrtf flname`	Write data from a file	dev, brd	7

Table 6-2. Syntax for NI-488.2 Routines in IBIC

Routine Syntax	Description	Note
`AllSpoll list`	Serial Poll all devices	11
`DevClear address`	Clear a device	13
`DevClearList list`	Clear several devices	11
`EnableLocal list`	Enable local control	11
`EnableRemote list`	Enable remote control	11
`FindLstn list limit`	Find all Listeners	11
`FindRQS list`	Find dev requesting service	11
`PassControl address`	Pass control to a device	13
`PPoll`	Parallel Poll	13
`PPollConfig addr. line sense`	Parallel Poll Configure	13,14
`PPollUnconfig address`	Parallel Poll Unconfig	13
`RcvRespMsg address data mode`	Receive Response Message	4,12,13
`ReadStatusByte address`	Serial Poll	13
`Receive address data mode`	Receive	4,12,13
`ReceiveSetup address`	Receive Setup	13
`ResetSys list`	3-level Device reset	11
`Send address data mode`	Send	4,12,13
`SendCmds data`	Send command bytes	4
`SendDataBytes list data mode`	Send Data Bytes	4,11,12
`SendIFC`	Send Interface Clear	
`SendList list data mode`	SendList	4,11,12
`SendLLO`	Put devices in LLO	11
`SendSetup list`	Send Setup	11
`SetRWLS list`	Put device in RWLS	11

(continues)

Table 6-2. Syntax for NI-488.2 Routines in IBIC (continued)

Routine Syntax	**Description**	**Note**
`TestSys list`	Device self-tests	11
`TestSRQ`	Test for SRQ	
`Trigger address`	Trigger a device	13
`TriggerList list`	Trigger several devices	11
`WaitSRQ`	Wait for SRQ	

Notes for Tables 6-1 and 6-2

1. `brdname` is the symbolic name of the new board (for example, `ibbna gpib1`).

2. Values enclosed in square brackets (`[]`) are optional. The default value is zero for `ibwait` and 1 for all other functions.

3. `v` is a hex, octal, or decimal integer. Hex numbers must be preceded by zero and x (for example, `0xD`). Octal numbers must be preceded by zero only (for example, `015`). Other numbers are assumed to be decimal.

4. `string` consists of a list of ASCII characters, octal or hex bytes, or special symbols. The entire sequence of characters must be enclosed in quotation marks. An octal byte consists of a backslash character followed by the octal value. For example, octal 40 would be represented by `\40`. A hex byte consists of a backslash character and a character `x` followed by the hex value. For example, hex 40 would be represented by `\x40`. The two special symbols are `\r` for a carriage return character and `\n` for a linefeed character. These symbols provide a more convenient method for inserting the carriage return and linefeed characters into the string, as shown in the following string: `"F3R5T1\r\n"`. Because the carriage return can be represented equally well in hex, `\xD` and `\r` are equivalent strings.

5. `udname` is the symbolic name of the new device or board (for example, `ibfind dev1` or `set gpib0`).

6. `v` is the number of bytes to read.

7. `flname` is the MS-DOS pathname of the file to be read or written (for example, `\test\meter` or `printr.buf`).

8. `mask` is a hex, octal, or decimal integer (see note 3) or a mask bit mnemonic.

9. `ibdev` parameters are `board id`, `pad`, `sad`, `tmo`, `eos`, and `eot`.

10. `ibln` parameters are `pad` and `sad`.

11. `list` is a comma separated list of address integers, optionally enclosed in parentheses. An empty list can be expressed by empty parentheses.

12. `mode` is a termination mode mnemonic or integer. Mnemonics are `NLend`, `NULLend` for Send-type operations, and `STOPend` for Receive-type operations.

13. `address` is an integer representing a GPIB address. If only a primary GPIB address is required, simply enter that integer. If a secondary address is also required, create an integer with the primary address in the low-order byte, and the secondary address in the high-order byte; for example, `pad` 3 and `sad` 61_{16} could be expressed as 0x6103.

14. `line` and `sense` are integers representing the data line to respond on and the sense of the response.

15. `mn` is a mnemonic for a configuration parameter, or the equivalent integer value. See the description of `ibconfig` in Chapter 5, *NI-488 Software Characteristics and Routines*, for the allowed mnemonics and their values.

Status Word

All IBIC functions return the status word ibsta in two forms: a hex value in square brackets and a list of mnemonics in parentheses.

Example:

```
dev1:  ibwrt "f2t3x"
[900](rqs cmpl)
COUNT: 5

dev1:
```

In this example, the status word shows that the device function write operation completed successfully and that dev1 is requesting service.

Table 6-3 lists the mnemonics of the status word. This is the same list that is given in Table 3-1.

Table 6-3. Status Word Layout

Mnemonics	Bit Pos.	Hex Value	Function Type	Description
ERR	15	8000	dev, brd	GPIB error
TIMO	14	4000	dev, brd	Time limit exceeded
END	13	2000	dev, brd	END or EOS detected
SRQI	12	1000	brd	SRQ interrupt received
RQS	11	800	dev	Device requesting service
EVENT	10	400	brd	A DTAS or DCAS event has occurred

(continues)

Table 6-3. Status Word Layout (continued)

Mnemonics	**Bit Pos.**	**Hex Value**	**Function Type**	**Description**
SPOLL	9	200	brd	The board has been serial polled by the Controller
CMPL	8	100	dev, brd	I/O completed
LOK	7	80	brd	Lockout State
REM	6	40	brd	Remote State
CIC	5	20	brd	Controller-In-Charge
ATN	4	10	brd	Attention is asserted
TACS	3	8	brd	Talker
LACS	2	4	brd	Listener
DTAS	1	2	brd	Device Trigger State
DCAS	0	1	brd	Device Clear State

Error Code

If an NI-488 function or NI-488.2 routine completes with an error, `IBIC` also displays the error mnemonic. The following example illustrates the result if an error condition occurs in the data transfer.

Example:

```
dev1:    ibwrt "f2t3x"
[8100] (err cmpl)
ERROR:  ENOL
COUNT:  1

dev1:
```

In this example, there are no Listeners; perhaps `dev1` is powered off.

Byte Count

When an I/O function completes, IBIC displays the actual number of bytes sent or received, regardless of the existence of an error condition.

Auxiliary Functions

Table 6-4 summarizes the auxiliary functions that `IBIC` supports.

Table 6-4. Auxiliary Functions That `IBIC` Supports

Description	Function Syntax	Note
Select active device or board	`set udname` or `set 488.2 n`	1,2 7
Display help information	`help [option]`	3
Repeat previous function	`!`	
Turn OFF display	`-`	
Turn ON display	`+`	
Execute function `n` times	`n* function`	4
Execute previous function `n` times	`n* !`	
Execute indirect file	`$ filename`	5
Display string on screen	`print string`	6
Exit or quit	`e`	
Exit or quit	`q`	

Notes for Table 6-4

1. `udname` is the symbolic name of the new device or board (for example, `ibfind dev1` or `set gpib0`).

2. Call `ibfind` initially to open each device or board.

3. If `option` is omitted, a menu of options appears.

4. Replace function with correct `IBIC` function syntax.

5. `filename` is the MS-DOS pathname of a file that contains `IBIC` functions to be executed.

6. `string` consists of a list of ASCII characters, octal or hex bytes, or special symbols. The entire sequence of characters must be enclosed in quotation marks. An octal byte consists of a backslash character followed by the octal value. For example, octal 40 would be represented by `\040`. A hex byte consists of a backslash character and a character `x` followed by the hex value. For example, hex 40 would be represented by `\x40`. The two special symbols are `\r` for a carriage return character and `\n` for a linefeed character. These symbols provide a more convenient method for inserting the carriage return and linefeed characters into the string as shown in this string: `"F3R5T1\r\n"`. Since the carriage return can be represented equally well in hex, `\xD` and `\r` are equivalent strings.

7. `n` indicates a board number (for example, `n`=1 for `gpib1`).

HELP (Display Help Information)

The `HELP` function gives causal information about `IBIC` and its functions to be displayed on the screen.

! (Repeat Previous Function)

The `!` function causes the most recent function executed to be repeated.

Example:

Screen Image	Comments

`gpib0: ibsic`
`[130]   ( cmpl cic atn )`
	Send Interface Clear

`gpib0:  !`
`[130]   ( cmpl cic atn )`
	Repeat `ibsic`

`gpib0:  !`
`[130]   ( cmpl cic atn )`
	Repeat `ibsic` again

- (Turn OFF Display)

The `-` function causes the NI-488 function output *not* to display on the screen. This function is useful when you want to repeat a NI-488 I/O function quickly without waiting for screen output to be displayed.

+ (Turn ON Display)

The + function causes the display to be restored.

The following example shows how the `-` and + functions are used. Twenty-four consecutive letters of the alphabet are read from a device using three `ibrd` calls.

Example:

```
dev1:  ibrd 8
[4100]  (end cmpl)
COUNT:  8
61 62 63 64 65 66 67 68   a b c d e f g h

dev1:  -

dev1:  ibrd 8

dev1:  +

dev1:  ibrd 8
[4100]  (end cmpl)
COUNT:  8
71 72 73 74 75 76 77 78   q r s t u v w x
```

`n*` (Repeat Function n Times)

The `n*` function repeats the execution of the specified function `n` times, where `n` is an integer. In the following example, the message `Hello` is sent to the printer five times.

Example:

```
printer:  5*ibwrt "Hello"
```

The function name can be replaced with the `!` function. Thus, if this example is done the following way, the word `Hello` will be sent 20 more times, then 10 more times.

Example:

```
printer:   20* !
printer:   10* !
```

Notice that the multiplier (`*`) does not become part of the function name; that is, `ibwrt "Hello"` is repeated 20 times, not `5* ibwrt "Hello"`.

$ (Execute Indirect File)

In the `$` function, an indirect file is a text file that contains `IBIC` functions. It is similar to an MS-DOS batch file and is created the same way. `$` reads the specified indirect file and executes the `IBIC` functions in sequence as if they were entered in that order from the keyboard. For example, entering:

```
gpib0:   $ usrfile
```

executes the `IBIC` functions listed in the file `usrfile`, and entering:

```
gpib0:    3*$ usrfile
```

repeats that operation three times.

The display mode, in effect before this function was executed, is restored afterward but may be changed by functions in the indirect file.

PRINT (Display the ASCII String)

The PRINT function can be used to echo a string to the screen.

Example:

```
dev1:  print "hello"
hello
dev1: print "and\r\n\x67\x6f\x6f\x64\x62\x79\x65"
and
goodbye
```

PRINT can be used to display comments from indirect files. The print strings will appear even if the display is suppressed with the - function.

The second PRINT example illustrates the use of hex values in IBIC strings.

E or Q (exit or quit)

The MS-DOS exit command or the IBIC function E or Q returns you to MS-DOS.

IBIC Sample Programs

Refer to Chapter 4 and Chapter 5 for a description of the programming steps that may be used to program a representative IEEE-488 instrument from your personal computer using the NI-488.2 software routines and NI-488 functions, respectively. The applications are written using IBIC commands.

NI-488.2 Routines

To set up IBIC for NI-488.2 calls, use the set command.

Example:

```
: set 488.2

488.2 (0):
```

Send the interface clear message (IFC) to all devices. This clears the bus. The user should check for ERR after each GPIB function call to be safe.

Example:

```
488.2 (0):  SendIFC
[0120] (cmpl cic)
```

Clear the device. The device is assumed to be on the GPIB bus at primary address 2.

Example:

```
488.2 (0):  DevClear 2
[0138] (cmpl cic atn tacs)
count: 1
```

Write the routine, range, and trigger source information to the device (a digital voltmeter).

Example:

```
488.2 (0):  Send 2 "F3R7T3" DABend
[0128] (cmpl cic tacs)
count: 6
```

Trigger the device.

Example:

```
488.2 (0):  Trigger 2
[0138] (cmpl cic atn tacs)
count: 1
```

Wait for the meter to request service (by asserting the SRQ bus line) and then read the meter's status byte.

Example:

```
488.2 (0):  WaitSRQ
[1138] (srqi cmpl cic atn tacs)
SRQ line is asserted

488.2 (0):  ReadStatusByte 2
[0174] (cmpl rem cic atn lacs)
 Poll:  2 => 0x0040  (decimal : 32)
```

Read the meter's data.

Example:

```
488.2 (0):  Receive 2 20 STOPend
[2164] (end cmpl rem cic lacs)
count: 20
0d 0a 4e 44 43 56 2d 30  . . N D C V - 0
30 30 2e 30 30 34 37 45  0 0 . 0 0 4 7 E
2b 30 0d 0a              + 0 . .
```

Return to the DOS operating system.

Example:

```
488.2 (0):  e
```

Device Functions

To communicate with a device, first find the device name that was given to the device in the `IBCONF` program.

Example:

```
:  ibfind dvm
id = 32005

DVM:
```

Clear the device. The user should check for ERR after each GPIB function call to be safe.

Example:

```
DVM:  ibclr
[0100] (cmpl)
```

Write the function, range, and trigger source instructions to the DVM.

Example:

```
DVM:  ibwrt "F3R7T3"
[0100] (cmpl)
count: 6
```

Trigger the device.

Example:

```
DVM:  ibtrg
[0100] (cmpl)
```

Wait for the DVM to request service or for a timeout; if the current timeout limit is too short, use `ibtmo` to change it.

Example:

```
DVM:  ibwait (TIMO RQS)
[800] (rqs)
```

Read the serial poll status byte. This serial poll status byte varies depending on the device used.

Example:

```
      ibrsp          DVM:
             [0100] (cmpl)
Poll: 0x40 (decimal : 64)
```

The read command displays the data on the screen both in hex values and their ASCII equivalents.

Example:

```
DVM:  ibrd 18
[0100] (cmpl)
count: 18

4E 44 43 56 20 30 30 30  N D C V   0 0 0
2E 30 30 34 37 45 28 30  . 0 0 4 7 E + 0
OA OA
```

Return to DOS.

Example:

```
DVM: e
```

Board Functions

The following pages show how to execute board functions only, not device functions. For most applications, board functions are not needed.

Begin by making the interface board the current board.

Example:

```
:  ibfind gpib0
id = 32006

GPIB0:
```

Send the interface clear message (IFC) to all devices. This clears the bus and asserts attention (ATN) on the bus. The user should check for ERR after each GPIB function call to be safe.

Example:

```
GPIB0:  ibsic
[0130] (cmpl cic atn)
```

Turn on the remote enable signal (REN).

Example:

```
GPIB0:  ibsre 1
[0130] (cmpl cic atn)
previous value: 0
```

Set up the addressing for the device to listen and the computer to talk. The question mark (`?`) character represents the unlisten (UNL) command. The `@` character represents the talk address of the GPIB board, which was calculated using the Multiline Interface Message chart in Appendix A. The GPIB board is at GPIB primary address 0. Moving across to the Talk address column, the appropriate ASCII character is an `@` character. In a similar manner the `!` character represents the listen address of the device which in this case is assumed to be at GPIB primary address 1. The Multiline Interface Message chart in Appendix A indicates that the listen address for a device at a primary address of 1 is an `!` character.

Example:

```
GPIB0:  ibcmd "@?!"
[0138] (cmpl cic atn tacs)
count: 3
```

Write the function, range, and trigger source instructions to the DVM. Be sure an error has not occurred before proceeding with the sample program.

Example:

```
GPIB0:   ibwrt "F3R7T3"
[0128] (cmpl cic tacs)
count: 6
```

Send the Group Execute Trigger message (GET) to trigger a measurement reading. The GET message is represented by the hex value 8.

Example:

```
GPIB0:   ibcmd "\x08"
[0138] (cmpl cic atn tacs)
count: 1
```

Wait for the DVM to set SRQ or for a timeout; if the current timeout limit is too short, use `ibtmo` to change it.

Example:

```
GPIB0:   ibwait (TIMO SRQI)
[1138] (srqi cmpl cic atn tacs)
```

Set up the device for a serial poll. The question mark (`?`) character represents the Unlisten (UNL) command. The space character ( ) represents the Controller's listen address. The hex value 18 represents the Serial Poll Enable function, while `A` represents the talk address of the device.

Example:

```
GPIB0:   ibcmd "? \x18A"
[1174] (srqi cmpl rem cic atn lacs)
count: 4
```

Read the status byte. The status byte returned may vary depending on the device used.

Example:

```
GPIB0:   ibrd 1
[0164] (cmpl rem cic lacs)
count: 1
50                P
```

Complete the serial poll by sending the Serial Poll Disable (SPD) message. The hex value 19 represents the serial poll disable function.

Example:

```
GPIB0:   ibcmd "\x19"
[0174] (cmpl rem cic atn lacs)
count: 1
```

Because the DVM and the NI-488 are still addressed to talk and to listen, the measurement can be read.

Example:

```
GPIB0:   ibrd 20
[2164] (end cmpl rem cic lacs)
0D 0A 4E 44 43 56 2D 30  • • N D C V - 0
30 30 2E 30 30 34 37 45  0 0 . 0 0 4 7 E
2B 30 0D 0A              + 0 • •
```

Return to the DOS operating system.

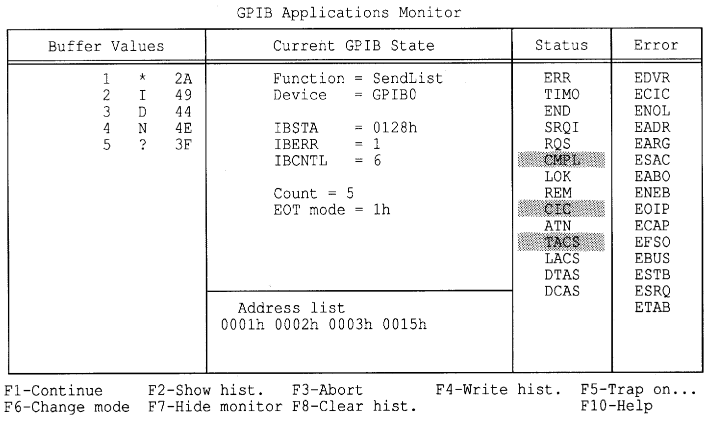
Example:

```
GPIB0: e
```

Chapter 7
Applications Monitor

This chapter explains how to use, install, and configure the applications monitor, a resident program that is useful in debugging sequences of GPIB calls from within your application. The applications monitor can halt program execution (trap) upon return from GPIB functions that meet a condition you specified. You then can inspect function arguments, buffers, return values, GPIB global variables, and other pertinent data. You may select the conditions that suspend the program. These conditions include the following: suspend on return from every NI-488.2 routine and NI-488 function, suspend on those functions that return an error indication, or suspend on those calls that are returned with selected bit patterns in the GPIB status word.

If the desired condition is met, you will see a pop-up screen (Figure 7-1) that contains details of the trapped function. In addition, you can view up to 255 of the preceding functions to verify that the sequence of calls and their arguments have occurred as intended.



```
                         GPIB Applications Monitor

 Buffer Values      | Current GPIB State         | Status | Error
--------------------+----------------------------+--------+------
        1   *   2A  | Function = SendList        | ERR    | EDVR
        2   I   49  | Device   = GPIB0           | TIMO   | ECIC
        3   D   44  |                            | END    | ENOL
        4   N   4E  | IBSTA    = 0128h           | SRQI   | EADR
        5   ?   3F  | IBERR    = 1               | RQS    | EARG
                    | IBCNTL   = 6               | CMPL   | ESAC
                    |                            | LOK    | EABO
                    | Count = 5                  | REM    | ENEB
                    | EOT mode = 1h              | CIC    | EOIP
                    |                            | ATN    | ECAP
                    |                            | TACS   | EFSO
                    |                            | LACS   | EBUS
                    |                            | DTAS   | ESTB
                    |                            | DCAS   | ESRQ
                    |  Address list              |        | ETAB
                    | 0001h 0002h 0003h 0015h    |        |

F1-Continue     F2-Show hist.   F3-Abort      F4-Write hist.  F5-Trap on...
F6-Change mode  F7-Hide monitor F8-Clear hist.                F10-Help
```

Figure 7-1. Applications Monitor Pop-up Screen

In many cases, you can omit explicit error-checking if you use the applications monitor. If a program is expected to run without errors, trapping on errors causes the applications monitor to be invoked only if an error occurs during a GPIB call. You may then take the action necessary to fix the problem.

Currently, the applications monitor is available with all revisions of the NI-488.2 for MS-DOS drivers.

Installing the Applications Monitor

The applications monitor is included on the distribution diskette as the file `APPMON.EXE`. To install it, type the following command in response to the DOS prompt:

```
APPMON
```

If the GPIB driver is not present or the applications monitor has already been installed, it does not load and an error message does print.

Once installed, the applications monitor remains in memory until you restart the system. Should you later decide that you no longer want to devote memory to the resident applications monitor, simply restart your system; the applications monitor will no longer be loaded.

IBTRAP

Once installed, the applications monitor is configured by the `ibtrap` function. The applications monitor can trap on GPIB driver calls that have certain bits set in the GPIB status word. The trap options are set by the special GPIB driver call, `ibtrap`. This call can be made either from the application program, or from the DOS prompt using the special utility program called `IBTRAP.EXE`.

With the function call and the DOS utility you select a *mask*, which determines those events that will be trapped, and a *monitor mode*, which selects what is displayed when a trap occurs.

The exact syntax of the function call is dependent on the language you are using. See the description of `ibtrap` in your language chapter for details about how to include `ibtrap` calls in your application.

The utility program `ibtrap` may be used to set the trap mode from DOS. Simply type `ibtrap` in response to the DOS prompt, specifying the desired combination of the flags listed on the following pages.

Select one or more *mask* flags:

`-all`	all GPIB calls
`-err`	GPIB error
`-timo`	timeout
`-end`	GPIB board detected END or EOS
`-srqi`	SRQ on
`-rqs`	device requesting service
`-event`	DTAS or DCAS messages are received
`-spoll`	GPIB board was serial polled
`-cmpl`	I/O completed
`-lok`	GPIB board is in Lockout State
`-rem`	GPIB board is in Remote State
`-cic`	GPIB board is Controller-In-Charge
`-atn`	attention is asserted
`-tacs`	GPIB board is Talker
`-lacs`	GPIB board is Listener
`-dtas`	GPIB board is in Device Trigger State
`-dcas`	GPIB board is in Device Clear State

Select only one *monitor* flag:

`-off`	turns the monitor off. No recording or trapping occurs.

`-rec`	instructs the monitor to record all GPIB driver calls but no trapping occurs.
`-dis`	instructs the monitor to record all GPIB driver calls and display whenever a trap condition exists.

Omitting either the *mask* or the *monitor* flags will leave the current configuration of `APPMON` unchanged. Invoking `ibtrap` without any flags displays the valid flags and their current state. This has no effect on the applications monitor configuration.

If you select the `-dis` option and you run an application program that places the PC's monitor into graphics mode, `APPMON` is not displayed. It does, however, continue recording events as if the `-rec` option had been used. When your PC's monitor is returned to character mode, `APPMON` is displayed on the next event.

By selecting various flags for the *mask* and *monitor* parameters, you may achieve a variety of trapping configurations. The following are some examples:

• `IBTRAP -cic -atn -dis`	Record all GPIB driver calls and display the applications monitor whenever attention is asserted or the GPIB board is Controller-in-Charge.
• `IBTRAP -srq -rec`	Record all GPIB driver calls and set the trap mask to trap when SRQ is on. Do not display the applications monitor when a trap condition exists.
• `IBTRAP -dis`	Record all GPIB driver calls and display the applications monitor whenever a trap condition exists. The trap mask remains unchanged.
• `IBTRAP -off`	Disable the applications monitor. No recording or trapping is performed.

See Chapter 5, *NI-488.2 Software Characteristics and Functions*, of this manual for the appropriate syntax to use in your application program.

Applications Monitor Options

When the applications monitor is displayed, you can view the parameters of the current GPIB call, change the display and trap modes, and scan the GPIB history. The applications monitor displays the following information pertinent to the current GPIB call:

• `Device`	symbolic device name.
• `Function`	NI-488.2 routine or NI-488 function description.
• `ibsta`	contains the GPIB status information.
• `iberr`	contains the GPIB error information or the previous value of the `value` parameter if no error occurred.
• `ibcntl`	contains a 32-bit representation of the number of bytes transferred.
• `Address List`	for functions that have an address list as a parameter, this displays the contents of the address list.
• `Buffer Value`	for functions that have a buffer as a parameter, this displays the contents of the buffer. Each byte of the buffer is shown with its index, character image, and ASCII value.
• `Status`	shows the state of the individual bits of `ibsta`. All active bits are highlighted.
• `Error`	shows the state of `iberr`. If an error occurred, the mnemonic for that error is highlighted.
• `Other`	additional parameters may be displayed, depending on the GPIB call.

Note: Numbers are in decimal unless immediately followed by the letter h, indicating that the number is in hex. The two exceptions (which are always in hex) are `ibsta` and the ASCII codes for buffer values. Also, the applications monitor is unable to record `ibfind` or `ibtrap` calls.

Main Commands

When the main applications monitor screen is displayed, the following command keys are available:

• <F1>	continue executing application program
• <F2>	display session summary
• <F3>	abort executing application program, return to DOS
• <F4>	write history to file
• <F5>	configure trap mask
• <F6>	configure monitor mode
• <F7>	hide/show monitor
• <F8>	clear GPIB history buffer
• <F10>	display command key list
• <Cursor Up>	scroll buffer up one character
• <Cursor Down>	scroll buffer down one character
• <Page Up>	scroll buffer up one page
• <Page Down>	scroll buffer down one page
• <Home>	scroll to beginning of buffer
• <End>	scroll to end of buffer
• <Cursor Right>	scroll address list(s) two characters to the right
• <Cursor Left>	scroll to beginning of address list(s)

GPIB History Screen

The GPIB history can be viewed by pressing F2. Once displayed, the following keys can manipulate the display:

• <Cursor Up>	scrolls history up one line
• <Cursor Down>	scrolls history down one line
• <Cursor Right>	scroll history two characters to the right
• <Cursor Left>	scroll history to the left-most position
• <Page Up>	scrolls history up one page
• <Page Down>	scrolls history down one page
• <Home>	scrolls to the top of history
• <End>	scrolls to the end of history
• <Escape> or <F2>	exits the GPIB history display and returns to the main applications monitor screen

Configuring the Trap Mask

Press F5 to change the current configuration of the trap mask. This action yields a pop-up menu with each of the status bits displayed along with their current state (either ON or OFF). Use the up and down arrow keys to highlight the desired bit and press <Spacebar> to toggle its state. Press <Enter> to record the changes. Pressing <Escape> cancels this action and leaves the mask unchanged. Selecting all bits has the effect of trapping on every call, while turning them all off causes no trapping to occur.

Configuring the Monitor Mode

Press F6 to change the current configuration of the applications monitor mode. This action yields a pop-up menu with the current mode selected. Use the up and down arrow keys to select the new mode and press <Enter> to record the change. Press <Escape> to cancel this action and leave the mode unchanged.

Hiding and Showing the Applications Monitor

Press F7 to hide the applications monitor and restore the contents of the screen. By pressing F7, you can view program output written to the screen, within the applications monitor, while the program is active. Pressing F7 again restores the applications monitor.

Exiting Directly to DOS

Press F3 to exit directly from your application back to DOS. This terminates your application and let you continue working from the DOS prompt.

Appendix A
Multiline Interface Messages

This appendix contains a multiline interface message reference list, which describes the mnemonics and messages that correspond to the interface functions. These multiline interface messages are sent and received with ATN TRUE.

For more information on these messages, refer to the ANSI/IEEE488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation*.

Multiline Interface Messages

Hex	Oct	Dec	ASCII	Msg	Hex	Oct	Dec	ASCII	Msg
00	000	0	NUL		20	040	32	SP	MLA0
01	001	1	SOH	GTL	21	041	33	!	MLA1
02	002	2	STX		22	042	34	"	MLA2
03	003	3	ETX		23	043	35	#	MLA3
04	004	4	EOT	SDC	24	044	36	$	MLA4
05	005	5	ENQ	PPC	25	045	37	%	MLA5
06	006	6	ACK		26	046	38	&	MLA6
07	007	7	BEL		27	047	39	'	MLA7
08	010	8	BS	GET	28	050	40	(	MLA8
09	011	9	HT	TCT	29	051	41	)	MLA9
0A	012	10	LF		2A	052	42	*	MLA10
0B	013	11	VT		2B	053	43	+	MLA11
0C	014	12	FF		2C	054	44	,	MLA12
0D	015	13	CR		2D	055	45	-	MLA13
0E	016	14	SO		2E	056	46	.	MLA14
0F	017	15	SI		2F	057	47	/	MLA15
10	020	16	DLE		30	060	48	0	MLA16
11	021	17	DC1	LLO	31	061	49	1	MLA17
12	022	18	DC2		32	062	50	2	MLA18
13	023	19	DC3		33	063	51	3	MLA19
14	024	20	DC4	DCL	34	064	52	4	MLA20
15	025	21	NAK	PPU	35	065	53	5	MLA21
16	026	22	SYN		36	066	54	6	MLA22
17	027	23	ETB		37	067	55	7	MLA23
18	030	24	CAN	SPE	38	070	56	8	MLA24
19	031	25	EM	SPD	39	071	57	9	MLA25
1A	032	26	SUB		3A	072	58	:	MLA26
1B	033	27	ESC		3B	073	59	;	MLA27
1C	034	28	FS		3C	074	60	<	MLA28
1D	035	29	GS		3D	075	61	=	MLA29
1E	036	30	RS		3E	076	62	>	MLA30
1F	037	31	US		3F	077	63	?	UNL

Message Definitions

DCL	Device Clear	MSA	My Secondary Address
GET	Group Execute Trigger	MTA	My Talk Address
GTL	Go To Local	PPC	Parallel Poll Configure
LLO	Local Lockout	PPD	Parallel Poll Disable
MLA	My Listen Address		

Multiline Interface Messages

Hex	Oct	Dec	ASCII	Msg	Hex	Oct	Dec	ASCII	Msg
40	100	64	@	MTA0	60	140	96	`	MSA0,PPE
41	101	65	A	MTA1	61	141	97	a	MSA1,PPE
42	102	66	B	MTA2	62	142	98	b	MSA2,PPE
43	103	67	C	MTA3	63	143	99	c	MSA3,PPE
44	104	68	D	MTA4	64	144	100	d	MSA4,PPE
45	105	69	E	MTA5	65	145	101	e	MSA5,PPE
46	106	70	F	MTA6	66	146	102	f	MSA6,PPE
47	107	71	G	MTA7	67	147	103	g	MSA7,PPE
48	110	72	H	MTA8	68	150	104	h	MSA8,PPE
49	111	73	I	MTA9	69	151	105	i	MSA9,PPE
4A	112	74	J	MTA10	6A	152	106	j	MSA10,PPE
4B	113	75	K	MTA11	6B	153	107	k	MSA11,PPE
4C	114	76	L	MTA12	6C	154	108	l	MSA12,PPE
4D	115	77	M	MTA13	6D	155	109	m	MSA13,PPE
4E	116	78	N	MTA14	6E	156	110	n	MSA14,PPE
4F	117	79	O	MTA15	6F	157	111	o	MSA15,PPE
50	120	80	P	MTA16	70	160	112	p	MSA16,PPD
51	121	81	Q	MTA17	71	161	113	q	MSA17,PPD
52	122	82	R	MTA18	72	162	114	r	MSA18,PPD
53	123	83	S	MTA19	73	163	115	s	MSA19,PPD
54	124	84	T	MTA20	74	164	116	t	MSA20,PPD
55	125	85	U	MTA21	75	165	117	u	MSA21,PPD
56	126	86	V	MTA22	76	166	118	v	MSA22,PPD
57	127	87	W	MTA23	77	167	119	w	MSA23,PPD
58	130	88	X	MTA24	78	170	120	x	MSA24,PPD
59	131	89	Y	MTA25	79	171	121	y	MSA25,PPD
5A	132	90	Z	MTA26	7A	172	122	z	MSA26,PPD
5B	133	91	[	MTA27	7B	173	123	{	MSA27,PPD
5C	134	92	\	MTA28	7C	174	124	|	MSA28,PPD
5D	135	93	]	MTA29	7D	175	125	}	MSA29,PPD
5E	136	94	^	MTA30	7E	176	126	~	MSA30,PPD
5F	137	95	_	UNT	7F	177	127	DEL	

PPE	Parallel Poll Enable	SPE	Serial Poll Enable
PPU	Parallel Poll Unconfigure	TCT	Take Control
SDC	Selected Device Clear	UNL	Unlisten
SPD	Serial Poll Disable	UNT	Untalk

Appendix B
Common Errors and Their Solutions

This appendix lists the most common errors and their solutions. They are arranged according to the error code that is returned from the function as indicated by `iberr`. This appendix also describes error situations that do not return an error code. A full explanation of all possible errors can be found in the *Error Variable-iberr* discussion in Chapter 3, *Understanding the NI-488.2 Software*.

EDVR(0)

Error Condition: DOS error (`IBCNT` represents the value of the DOS error code).

Solutions:

- Check that `CONFIG.SYS` contains the line

  ```
  device=y:\dir\gpib.com
  ```

  where `y` is the startup drive and *`dir`* is the directory name where `INSTALL` copied the NI-488.2 software files; for example, if you are using the AT-GPIB, *`dir`* may be `AT-GPIB`.

  Enter `TYPE \CONFIG.SYS` from the startup drive and verify that the line exists there. Also verify that the file `GPIB.COM` exists in the above directory.

- Restart your system after you install the software.

- Using `IBCONF`, verify that the GPIB board and device names are set the way you expect.

ECIC(1)

Error Condition: Function requires GPIB board to be Controller-In-Charge.

Solutions:

- Run `IBCONF` and make sure the board being used (`GPIB0` or `GPIB1`) is configured to be the System Controller.
- If executing board functions, call `ibsic` to become Controller-In-Charge before any other function calls that require that capability.
- If control has been passed with an `ibpct` call, wait for it to be returned with the `ibwait` function.

ENOL(2)

Error Condition: Function detected no Listeners.

Solutions:

- Check that the device is powered on and also that at least two-thirds of the devices on the GPIB are turned on.
- Inspect the interconnecting cable to see that the devices are attached and that the connectors are seated properly.
- Check the switches or control panel of the device and verify that it has the correct GPIB address. Also check if the device uses extended addressing and requires a primary and secondary address. (Some devices use multiple secondary addresses to enable different internal functions).
- For device write functions, run `IBCONF` and check that the address of the device (including the secondary address) is correct. If a change is made, restart the system. Then run `IBIC` to verify that the address is correct, as follows. Open the device you want to write to using `ibfind` and execute `ibpad` and `ibsad` calls, passing each the address value you believe is correct. If secondary addressing is not used, pass a value of zero to the `ibsad` function. Assuming these calls do not return with an error, `IBIC` returns the previous address value,

which is the same as the new one if you have correctly configured the device.

- For board write functions, make sure the device is addressed properly using the `ibcmd` function before the write call. Verify that the low five bits of the listen address (and, if appropriate, the secondary address) used in the `ibcmd` call match the GPIB address(es) of the device and also that the listen address is in the range 20 through 3E hex (32 through 62 decimal) and the secondary address is in the range 60 through 7E hex (92 through 126 decimal).

EADR(3)

Error Condition: GPIB board (`GPIB0` or `GPIB1`) is not addressed correctly.

Solutions:

- Use `ibcmd` to send the appropriate Talk or Listen address before attempting an `ibwrt` or `ibrd`.
- If calling `ibgts` with the shadow handshake feature, call `ibcac` to ensure that the GPIB ATN line is asserted.

EARG(4)

Error Condition: Invalid argument to function call.

Solutions:

- Errors received from `IBIC`:
  - Verify syntax in Chapter 6, *IBIC*, of this manual.
  - Make sure the GPIB address of the board in `IBCONF` does not conflict with that of a device.

- Errors received when running your application program:
  - Verify syntax in Chapters 4 and 5 of this manual.
  - Make sure the GPIB address of the board in `IBCONF` does not conflict with that of a device.

ESAC(5)

Error Condition: GPIB board not System Controller as required.

Solutions:

- Run `IBCONF` and make sure the board (`GPIB0` or `GPIB1`) is configured to be System Controller.
- Issue a board `ibrsc` function call with a value of 1 to request System Control.

EABO(6)

Error Condition: I/O operation aborted.

Solutions:

- Check that the device is powered on.
- Verify proper cable connections.
- Errors received from `ibrd`:
  - Some devices do not send data unless they have received data telling them what to send. This is caused by devices having the capability of sending several types of data. Issue an `ibwrt` to set up the device and then an `ibrd` to receive the information.
  - If you have not changed any of the default EOS or EOI settings in `IBCONF`, the reads terminate when the buffer is full or when EOI is set. If your device sends an EOS termination character such as a carriage return rather than EOI, then use `IBCONF` to change the device characteristics.

ENEB(7)

Error Condition: Non-existent GPIB board.

Solution:

- Run `IBCONF` and make sure the base I/O address of the board matches the hardware address switch settings. `IBCONF` will display the proper switch settings for the selected base address.

EDMA(8)

Error Condition: Windows 3 only. Virtual DMA device error.

Solution:

- Refer to *Using Your NI-488.2 Software with Microsoft Windows* for details on how to use the replacement DMA device if you are running version 3.0 of Windows with DMA enabled. If you are running version 3.1 of Windows or version 3.0 of Windows with `nivdmad.386`, restart Windows to return the virtual DMA device to a known state.

EOIP(10)

Error Condition: Function not allowed while asynchronous I/O is in progress.

Solution:

- When using asynchronous functions, call `ibwait` to wait for CMPL (or TIMO CMPL) before making another call. Refer to the description of the asynchronous I/O function in Chapter 5, *NI-488 Software Characteristics and Functions*, for more details.

ECAP(11)

Error Condition: No capability for operation.

Solution:

- Run `IBCONF` and verify that the capability to do a particular call is enabled (for example, the interface board must be configured as System

Controller to execute the `ibsre` function). Check both device and board capabilities. Restart after leaving `IBCONF` if you made any changes.

EFSO(12)

Error Condition: File system error.

Solutions:

- Check the disk files to make sure names are properly specified and that the file exists.
- If more room is needed on the disk, delete some files.
- Rename any files which have the same name as a device named in `IBCONF`, or rename the device.

EBUS(14)

Error Condition: Command byte transfer error.

Solutions:

- Find out which device is abnormally slow to accept commands and fix the problem with the device.
- If more time is needed to send commands, lengthen the time limit in `IBCONF` or with `ibtmo`.

ESTB(15)

Error Condition: Serial poll status byte(s) lost.

Solutions:

- Call `ibrsp` more often to read the status bytes.
- Ignore ESTB.

ESRQ(16)

Error Condition: SRQ stuck in the ON position.

Solutions:

- Ignore ESRQ until all devices are found. ESRQ occurs because the device asserting SRQ was not opened with `ibfind`. The automatic serial polling function polls only the opened devices.
- Check that you have used `ibfind` to open all devices on the GPIB that could assert SRQ. Remove any device from the bus that is not being accessed.
- Using `IBIC`, attach one device at a time and determine that it is unasserting SRQ after being polled.
- Inspect the interconnecting cable to see that the devices are attached and that the connectors are seated properly.

ETAB(20)

Error condition: Table problem.

Solutions:

- For the `FindLstn` routine, this is a warning and not an error condition. You can either ignore this message or increase the size of the buffer.
- For the `FindRQS` routine, this error indicates that none of the specified devices are requesting services. Check to be sure that the device list contains the addresses of all on line devices. Also, `FindRQS` should normally be called only when the SRQ line is asserted. Use the `TestSRQ` or `WaitSRQ` routines to determine the state of SRQ.
- For the `ibevent` function, this error indicates that too many device clear or group execute trigger messages were received. The most recent messages will be lost. Try calling `ibevent` more often to keep the event queue empty.

Other Error Conditions

Listed below are general errors which may occur when using the GPIB hardware and software.

Error Condition: Attempts to run the GPIB utility programs, `ibdiag`, `ibtest`, or `IBIC` results in `Bad command or Filename` error from DOS. Also, the distribution files do not appear to be on the startup disk after running `INSTALL`.

Solution:

- `INSTALL` copies the files to a predefined subdirectory on the startup disk. This directory is by default

  ```
  y:\dir
  ```

  where `y` is the letter name of your startup disk and `dir` is the name of your GPIB board (for example: `c:\AT-GPIB`).

  Change to the predefined directory with the DOS command `cd` to run other programs.

Error Condition: Function call does not return, and the program seems to hang forever.

Solutions:

- Run `IBCONF` and confirm that the DMA channel and interrupt line of the board match the hardware settings. Then run `ibtest` after restarting. Also check that the DMA channel and interrupt line do not conflict with other devices in the computer.

- Check that the time limit is not set to zero, which results in infinite time limits. To do this, run `IBCONF` and look at the time limit value both for your device and your board. Or, from `IBIC`, call `ibtmo` and pass to it the timeout argument value you believe is correct (the default is 13, which represents 10 sec). Assuming that the `ibtmo` call does not result in an error, the previous value returned by the call equals the value passed to it if the time limit was what you expected.

- Check that you do not call `ibwait` for an event that may not happen without also waiting for TIMO as well. For example, if you fail to set the TIMO bit while waiting for RQS from a device, and for some reason the device does not assert SRQ, the `ibwait` call does not return.

Error Condition: The computer crashes.

Solutions:

- Run `IBCONF` and confirm that the DMA channel and interrupt line of the board match the hardware settings. Also check that the DMA channel and interrupt line do not conflict with other devices in the computer.
- Configure the hardware and software to not use DMA and/or interrupts. You may have a PC compatible that is not *totally* compatible.
- Check that none of your device names is the same as any of your file or directory names, not including the file or directory name suffix.

Error Condition: `ibdiag` reports a DMA or interrupt problem.

Solutions:

- Reconfigure the hardware and software for another DMA channel and/or interrupt line. You might have a conflict with another device.
- Check that another device on the same line really has the shareable interrupt capability. Many do not.
- If an interrupt or DMA problem persists, configure the hardware and software for non-DMA or non-interrupt operation. You may have a PC compatible that is not *totally* compatible.

Error Condition: `IBIC` reports an error on `ibfind` or `ibfind` returns a negative unit descriptor.

Solutions:

- Check the solutions listed for EDVR, ENEB, or EARG error codes.
- Run `IBCONF` to check that the device or board name is correct.

Error Condition: The GPIB device being programmed appears to accept the instruction but does not execute properly.

Solutions:

- Check that the instruction sent with the write function contains the proper delimiters and message termination characters. For example, some devices require a carriage return and/or a linefeed character before they execute the instruction. Other devices require special characters to separate multiple instruction messages. Your instruction manual may be incomplete or ambiguous, so try several possible combinations.
- When running a program, check for errors after each NI-488 function call or NI-488.2 routine. If an early call failed but you did not check `IBSTA`, later functions will execute improperly and give misleading status data.
- Check that the board you are using and the GPIB device you are programming are not at the same GPIB address. If using device functions, run `IBCONF` to perform the check.

Appendix C
Serial/Parallel Port Redirection to the GPIB

This appendix describes how to use standard DOS printer commands (LPRINT, PRINT, and so on) to redirect data from the serial or parallel port to a GPIB printer, plotter, or other device using the NI-488.2 for MS-DOS driver. This appendix explains how to do this.

How to Redirect Data

When installing the NI-488.2 software as described in Chapter 2, *Installation and Configuration of the NI-488.2 Software*, run the configuration program `IBCONF` to change the device name of your GPIB printer and plotter to the DOS printer command. This action should cause DOS to bypass the standard printer device and use the NI-488.2 driver to write to the GPIB device instead.

Note: The redirected data is sent in a unidirectional manner by intercepting DOS interrupts. A software package that expects bidirectional communication with the GPIB device or that does not generate DOS interrupts when outputting to the serial or parallel ports *will not* do this redirection.

The following procedure and examples demonstrate the use of parallel port redirection (`LPT1`) but could also be used for redirection to other ports (`COM1`, `LPT2`, `LPT3`).

To install the software follow these steps:

1. Run `INSTALL` as explained in Chapter 2.

2. Run `IBCONF` and follow all instructions.

3. In `IBCONF`

    a. Change the name `dev1` to `LPT1`.

   b. Change the primary address of `LPT1` to the GPIB address of your printer.

4. Exit `IBCONF`.

5. Restart the system.

6. Run `ibtest` to verify correct software installation.

Examples:

System

```
PRINT FILE
COPY FILE LPT1
```

BASIC

```
10  LPRINT "hello"

10  OPEN "LPT1" FOR OUTPUT AS #1
20  PRINT #1, "it works!"
```

Appendix D
Operation of the GPIB

This appendix describes some basic concepts you should understand to operate the GPIB. It also contains a description of the physical and electrical characteristics as well as configuration requirements of the GPIB.

Types of Messages

The GPIB carries device-dependent messages and interface messages.

- Device-dependent messages, often called *data* or *data messages*, contain device-specific information such as programming instructions, measurement results, machine status, and data files.
- Interface messages manage the bus itself. They are usually called *commands* or *command messages*. Interface messages perform such tasks as initializing the bus, addressing and unaddressing devices, and setting device modes for remote or local programming.

The term *command* as used here should not be confused with some device instructions which can also be called commands. Such device-specific instructions are actually data messages.

Talkers, Listeners, and Controllers

A *Talker* sends data messages to one or more *Listeners*. The *Controller* manages the flow of information on the GPIB by sending commands to all devices.

Devices can be Listeners, Talkers, and/or Controllers. A digital voltmeter, for example, is a Talker and may be a Listener as well.

The GPIB is a bus like an ordinary computer bus, except that the computer has its circuit cards interconnected via a backplane bus, whereas the GPIB has stand-alone devices interconnected via a cable bus.

The role of the GPIB Controller can also be compared to the role of the CPU of a computer, but a better analogy is to the switching center of a city telephone system.

The switching center (Controller) monitors the communications network (GPIB). When the center (Controller) notices that a party (device) wants to make a call (send a data message), it connects the caller (Talker) to the receiver (Listener).

The Controller addresses a Talker and a Listener before the Talker can send its message to the Listener. After the message is transmitted, the Controller may unaddress both devices.

Some bus configurations do not require a Controller. For example, one device may always be a Talker (called a Talk-only device) and there may be one or more Listen-only devices.

A Controller is necessary when the active or addressed Talker or Listener must be changed. The Controller function is usually handled by a computer.

With the GPIB interface board and its software your personal computer plays all three roles.

- Controller - to manage the GPIB
- Talker - to send data
- Listener - to receive data

The Controller-In-Charge and System Controller

Although there can be multiple Controllers on the GPIB, only one Controller at a time is active or Controller-In-Charge (CIC). Active control can be passed from the current CIC to an idle Controller. Only one device on the bus, the System Controller, can make itself the CIC. The GPIB interface board is usually the System Controller.

GPIB Signals and Lines

The interface system consists of 16 signal lines and 8 ground return or shield drain lines.

The 16 signal lines are divided into the following three groups.

- Eight data lines
- Three handshake lines
- Five interface management lines

Data Lines

The eight data lines, DI01 through DI08, carry both data and command messages. All commands and most data use the 7-bit ASCII or ISO code set, in which case the eighth bit, DI08, is unused or used for parity.

Handshake Lines

Three lines asynchronously control the transfer of message bytes among devices. The process is called a three-wire interlocked handshake, and it guarantees that message bytes on the data lines are sent and received without transmission error.

NRFD (not ready for data)

NRFD indicates when a device is ready or not ready to receive a message byte. The line is driven by all devices when receiving commands and by Listeners when receiving data messages.

NDAC (not data accepted)

NDAC indicates when a device has or has not accepted a message byte. The line is driven by all devices when receiving commands and by Listeners when receiving data messages.

DAV (data valid)

DAV tells when the signals on the data lines are stable (valid) and can be accepted safely by devices. The Controller drives DAV when sending commands and the Talker drives it when sending data messages.

Interface Management Lines

Five lines are used to manage the flow of information across the interface.

ATN (attention)

The Controller drives ATN true when it uses the data lines to send commands and false when it allows a Talker to send data messages.

IFC (interface clear)

The System Controller drives the IFC line to initialize the bus and become CIC.

REN (remote enable)

The System Controller drives the REN line, which is used to place devices in remote or local program mode.

SRQ (service request)

Any device can drive the SRQ line to asynchronously request service from the Controller with the SRQ line.

EOI (end or identify)

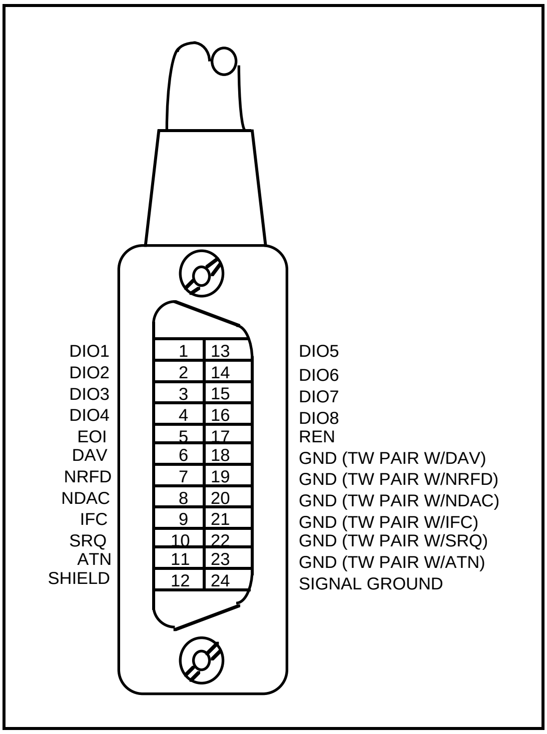
The EOI line has two purposes. The Talker uses the EOI line to mark the end of a message string. The Controller uses the EOI line to tell devices to identify their response in a parallel poll.

Physical and Electrical Characteristics

Devices are usually connected with a cable assembly consisting of a shielded 24 conductor cable with both a plug and receptacle connector at each end. This design allows devices to be linked in either a linear or a star configuration, or a combination of the two. See Figures D-1, D-2, and D-3.

The standard connector is the Amphenol or Cinch Series 57 *Microribbon* or *Amp Champ* type. An adapter cable using a non-standard cable and/or connector is used for special interconnection applications.

The GPIB uses negative logic with standard TTL logic level. When DAV is true, for example, it is a TTL low level (≤ 0.8V), and when DAV is false, it is a TTL high level (≥ 2.0V).



DIO1 1 13 DIO5
DIO2 2 14 DIO6
DIO3 3 15 DIO7
DIO4 4 16 DIO8
EOI 5 17 REN
DAV 6 18 GND (TW PAIR W/DAV)
NRFD 7 19 GND (TW PAIR W/NRFD)
NDAC 8 20 GND (TW PAIR W/NDAC)
IFC 9 21 GND (TW PAIR W/IFC)
SRQ 10 22 GND (TW PAIR W/SRQ)
ATN 11 23 GND (TW PAIR W/ATN)
SHIELD 12 24 SIGNAL GROUND

Figure D-1. GPIB Connector and the Signal Assignment

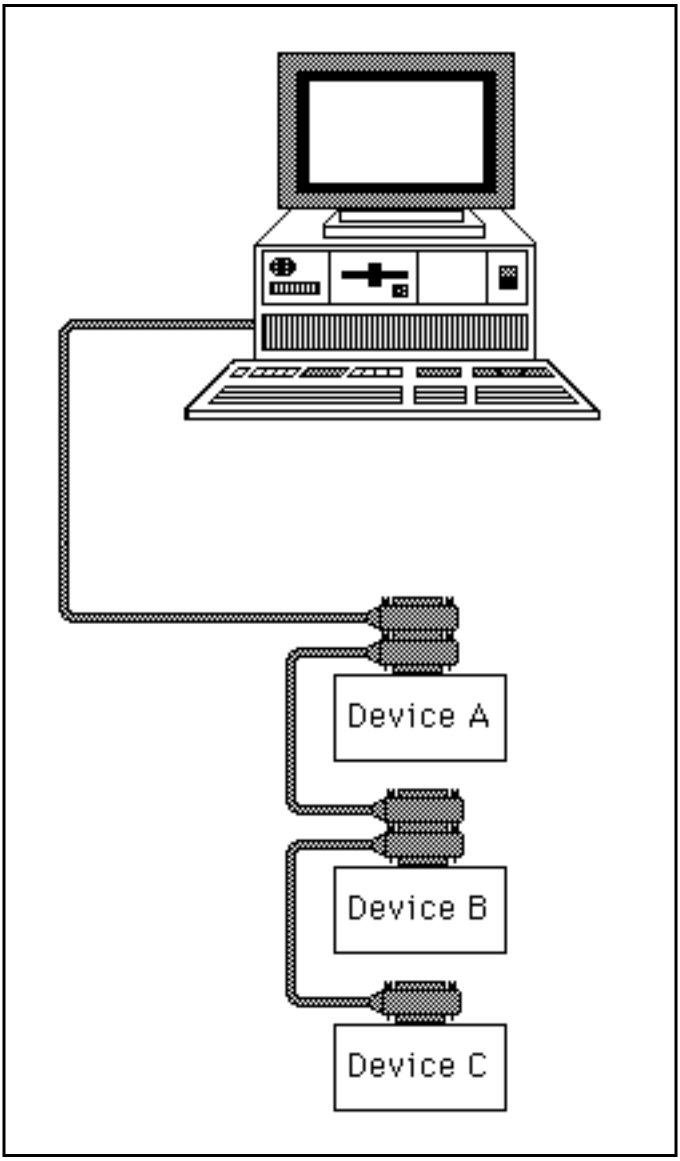


Figure D-2. Linear Configuration

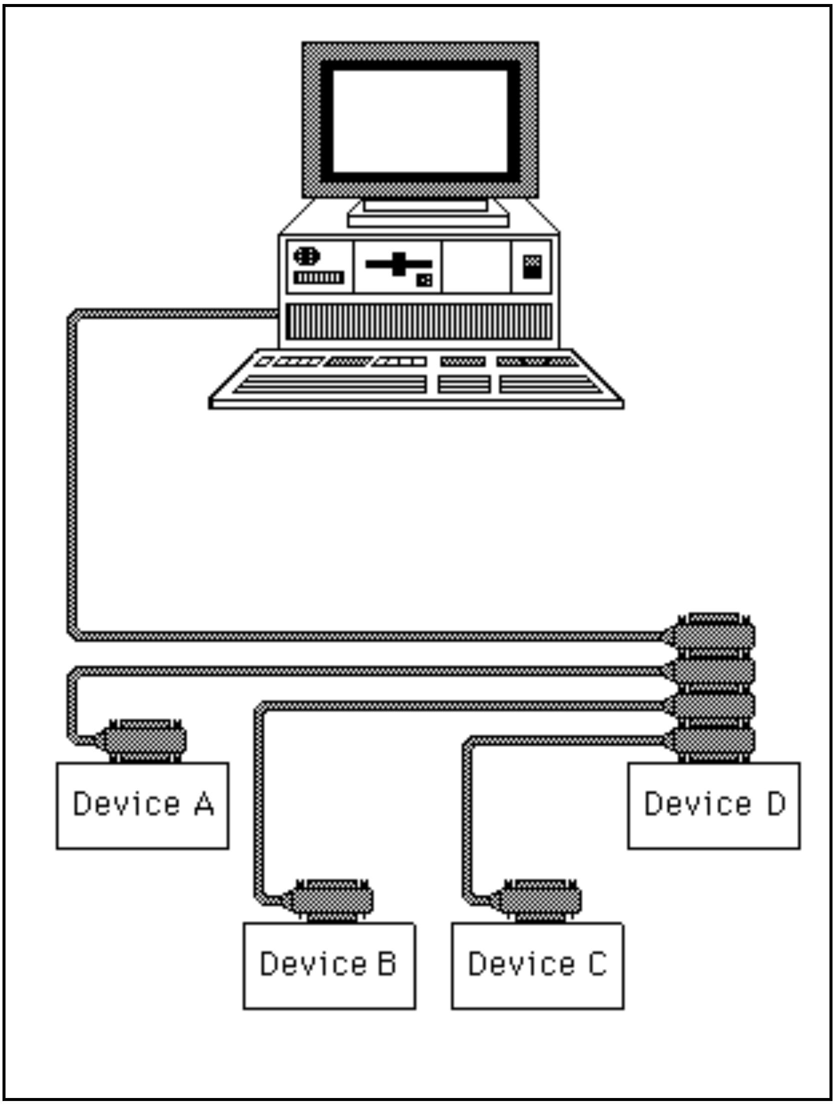


Figure D-3. Star Configuration

Configuration Requirements

To achieve the high data transfer rate that the GPIB was designed for, the physical distance between devices and the number of devices on the bus are limited.

The following restrictions are typical.

- A maximum separation of four meters between any two devices and an average separation of two meters over the entire bus.
- A maximum total cable length of 20 meters.
- No more than 15 devices connected to each bus, with at least two-thirds powered on.

Bus extenders are available from National Instruments for use when these limits must be exceeded.

Related Document

For more information on topics covered in this section, consult the ANSI/IEEE488.1-1987 Standard, *IEEE Standard Digital Interface for Programmable Instrumentation.*

Appendix E
Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a Fax-on-Demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by application engineers.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call (512) 795-6990. You can access these services at:

United States: (512) 794-5422

Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422

Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59

Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at (512) 418-1111.

E-Mail Support (currently U.S. only)

You can submit technical support questions to the applications engineering team through e-mail at the following Internet address. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions:
`support@natinst.com`

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

	Telephone	**Fax**
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 5734815	03 5734816
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name ____

Title ____

Company ____

Address ____

____

Fax (___) ______________________ Phone (___) __________________

Computer brand ________ Model _______________ Processor ___________

Operating system (include version number) ____________________________

Clock Speed _______Mhz RAM _________ MB Display adapter ________

Mouse _____yes _____ no Other adapters installed _______________________

Hard disk capacity _______ MB Brand __________________________________

Instruments used ____

National Instruments hardware product model _____________Revision ______

Configuration____

National Instruments software product ____________________ Version______

Configuration____

The problem is____

____

____

____

____

List any error messages ____

____

The following steps will reproduce the problem___________________________

____

____

____

NI-488.2 Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

[The information below is product-specific. Actual contents vary according to product. Check with your content expert and product manager.]

National Instruments Products

- NI-488.2 Software Revision/Version Number on Distribution Medium: ____________________________________

- Application Programming Language (BASICA, QuickBASIC, C, Pascal, and so on): ____

- Programming Language Interface Revision:___________________________

- Type of National Instruments GPIB boards installed and their respective hardware settings:

Board Type	Interrupt Level	DMA Channel	Base I/O Address

(continues)

Other Products

- Computer Make and Model: ______________________________
- Microprocessor: ____
- Clock Frequency: _______________________________________
- Type of Monitor Card Installed: ___________________________
- DOS Version : ____
- Programming Language/Version: ___________________________
- Other Boards in System: ________________________________
- Base I/O Address of Other Boards: _________________________
- DMA Channels of Other Boards: ___________________________
- Interrupt Level of Other Boards: ___________________________

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title:	**NI-488.2™ Software Reference Manual for MS-DOS**
Edition Date:	**June 1997**
Part Number:	**370889A-01**

Please comment on the completeness, clarity, and organization of the manual.

(continues)

If you find errors in the manual, please record the page numbers and describe the errors.

____

____

____

____

____

____

____

____

____

____

____

Thank you for your help.

Name ____

Title ____

Company ____

Address ____

____

Phone (_____) _________________

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway, MS 53-02
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678

Glossary

A

acceptor handshake	A GPIB interface function that receives data or commands. Listeners use this function to receive data, and all devices use it to receive commands. See *source handshake* and *handshake*.
access board	The GPIB board that controls and communicates with the devices on the bus that are attached to it.
applications monitor	A resident program that is useful in debugging sequences of GPIB calls from within your application.
`appmon.exe`	The file on the distribution diskette that contains the applications monitor.
ATN or attention	A GPIB line that distinguishes between commands and data messages. When ATN is asserted, bytes on the GPIB DIO lines are commands.
automatic serial polling	A feature of the NI-488 software in which serial polls are executed automatically by the driver whenever a device asserts the GPIB SRQ line.

B

board	One of the GPIB interface boards in the computer. See *device*.
board function	A function that operates on or otherwise pertains to one of the GPIB interface boards in the computer. These boards are referred to as `GPIB0`, `GPIB1`, and so on. See *device function*.

C

CIC	See *Controller*.
command or command message	Common term for interface message.
`config.sys`	The DOS file that contains the names of the loadable device drivers that DOS loads when it is started up.
configuration	The process of altering the software parameters in the driver that describe the key characteristics of the devices and boards that are manipulated by the driver. By keeping this information, such as GPIB address, in the driver, it does not have to be defined in each application program. `ibconf.exe` is the NI-488 configuration program.
Controller or Controller-In-Charge	The device that manages the GPIB by sending interface messages to other devices.

D

data or data message	Common term for device-dependent message.
DAV or data valid	One of the three GPIB handshake lines. See *handshake*.
DCL or device clear	The GPIB command used to reset the device or internal functions of all devices. See *IFC* and *SDC*.
declaration file	An NI-488 file that contains code that must be placed at the beginning of an application program to allow it to properly access the driver. `DECL.BAS` is the declaration file for programs written in Interpretive BASICA. See *language interface*.

device	An instrument, peripheral, computer, or other electronics equipment that can be programmed over the GPIB. See *board.*
device clear	See *DCL.*
device-dependent message	A message sent from one device to another device, such as programming instructions, data, or device status. See *command* or *interface message*.
device function	A function that operates on or otherwise pertains to a GPIB device rather than to the GPIB interface board in the computer. See *board function.*
DI01 through DI08	The GPIB lines that are used to transmit command or data bytes from one device to another.
DMA or direct memory access	High-speed data transfer between the GPIB and memory that is not handled directly by the CPU. Not available on some systems. See *programmed I/O*.
driver	Common term for software used to manipulate a device or interface board.

E

END or end message	A message that signals the end of a data string. `END` is sent by asserting the GPIB End Or Identify (EOI) line with the last data byte.
EOI	A GPIB line that is used to signal either the last byte of a data message (END) or the parallel poll identify (IDY) message.
EOS or EOS byte	A 7- or 8-bit End-of-String character that is sent as the last byte of a data message.

G

General Purpose Interface Bus	See *GPIB*.
GET or group execute trigger	The GPIB command used to trigger a device or internal function of an addressed Listener.
go to local	See *GTL*.
GPIB	General Purpose Interface Bus is the common name for the communications interface system defined in ANSI/IEEE488.1-1987 Standard. Hewlett-Packard, the inventor of the bus, calls it the HP-IB.
GPIB address	The address of a device on the GPIB, composed of a primary address (MLA and MTA) and perhaps a secondary address (MSA). The GPIB board has both a GPIB address and an I/O address.
GPIB board	Reference to the National Instruments family of GPIB interface boards.
`gpib.com`	The NI-488.2 driver filename.
group execute trigger	See *GET*.
GTL or go to local	The GPIB command used to place an addressed Listener in local (front panel) control mode.

H

handshake	The mechanism used to transfer bytes from the source handshake function of one device to the acceptor handshake function of another device. The three GPIB lines DAV, NRFD, and NDAC are used in an interlocked fashion to signal the phases of the transfer, so that bytes can be sent asynchronously (for example, without a clock) at the speed of the slowest device.

hard startup	See *startup.*
high-level function	A device function that combines several rudimentary board operations into one function so that the user does not have to be concerned with bus management or other GPIB protocol matters. See *low-level function.*

I

`ibcnt` and `ibcntl`	Global variables that are updated after each I/O function call to show the actual number of bytes sent or received. `ibcnt` is a 16-bit representation of the transfer amount, while `ibcntl` is the full 32-bit representation.
`ibconf.exe`	The NI-488.2 driver configuration program. See *configuration.*
`iberr`	A global variable that contains the specific error code associated with a function call that failed.
`ibic`	The Interface Bus Interactive Control program is used to communicate with GPIB devices, troubleshoot problems, and develop your application.
`ibsta`	A global variable that is updated at the end of each function call with important status information such as the occurrence of an error.
`install.exe`	The NI-488.2 software installation program.
`ibtest.exe`	The NI-488.2 software diagnostic program.
IFC or interface clear	A GPIB line used by the System Controller to initialize the bus. See *DCL* and *SDC.*
interface message	A broadcast message sent from the Controller to all devices and used to manage the GPIB. Common interface messages include Interface

	Clear, listen addresses, talk addresses, and Serial Poll Enable/Disable. See *data* or *device-dependent message*.
I/O or input/output	In the context of this manual, the transmission of commands or messages between the computer via the GPIB board and other devices on the GPIB.
I/O address	The address of the GPIB board from the point of view of the CPU, as opposed to the GPIB address of the GPIB board. Also called port address or board address.
ist	An individual status bit of the status byte used in the parallel poll configure function.

L

LAD or listen address	See *MLA*.
language interface	Code that enables an application program written in a particular language to call handler functions. `BIB.M` is the language interface for Interpretive BASIC.
listen address	See *MLA*.
Listener	A GPIB device that receives data messages from a Talker.
LLO or local lockout	The GPIB command used to tell all devices that they may or should ignore remote (GPIB) data messages or local (front panel) controls, depending on whether the device is in local or remote program mode.
low-level function	A rudimentary board or device function that performs a single operation. See *high-level function*.

M

MLA or my last address	The GPIB command used to address a device to be a Listener. There are 31 of these primary addresses.
MSA or my secondary address	The GPIB command used to address a device to be a Listener or a Talker when extended (two byte) addressing is used. The complete address is a MLA or MTA address followed by an MSA address. There are 31 of these secondary addresses for a total of 961 distinct listen or talk addresses for devices.
MTA or my talk address	A GPIB command used to address a device to be a Talker. There are 31 of these primary addresses.

N

NDAC or not data accepted	One of the three GPIB handshake lines. See *handshake*.
NRFD or not ready for data	One of the three GPIB handshake lines. See *handshake*.

O

on pen instrument	National Instruments uses this statement to intercept SRQ interrupts and make them available to user programs.
opened device or board	A device or board that has been enabled or placed online by the `ibfind` function.

P

parallel poll	The process of polling all configured devices at once and reading a composite poll response. See *serial poll*.

parallel poll configure	See *PPC*.
parallel poll disable	See *PPD*.
parallel poll enable	See *PPE*.
parallel poll unconfigure	See *PPU*.
port address	See *I/O address*.
PPC or parallel poll configure	The GPIB command used to configure an addressed Listener to participate in polls.
PPD or parallel poll disable	The GPIB command used to disable a configured device from participating in polls. There are 16 PPD commands.
PPE or parallel poll enable	The GPIB command used to enable a configured device to participate in polls and to assign a DIO response line. There are 16 `PPE` commands.
PPU or parallel poll unconfigure	The GPIB command used to disable any device from participating in polls.
programmed I/O	Low-speed data transfer between the GPIB board and memory in which the CPU moves each data byte according to program instructions. See *DMA*.

R

REN or remote enable	A GPIB line controlled by the System Controller but used by the CIC to place devices in remote program mode.
root directory	The top-level directory on a diskette or hard disk.

S

SDC or selected device clear	The GPIB command used to reset internal or device functions of an addressed Listener. See *DCL* and *IFC*.
serial poll	The process of polling and reading the status byte of one device at a time. See *parallel poll*.
serial poll disable	See *SPD*.
serial poll enable	See *SPE*.
service request	See *SRQ*.
soft startup	See *startup*.
source handshake	The GPIB interface function that transmits data and commands. Talkers use this function to send data, and the Controller uses it to send commands. See *acceptor handshake* and *handshake*.
SPD or serial poll disable	The GPIB command used to cancel an `SPE` command.
SPE or serial poll enable	The GPIB command used to enable a specific device to be polled. That device must also be addressed to talk. See *SPD*.
SRQ or service request	The GPIB line that a device asserts to notify the CIC that the device needs servicing.
startup	To load the operating system programs from diskette or hard disk into memory and to begin executing the code. A hard startup is when power is applied to the computer. A warm or soft startup is when specific keys are pressed, such as <Ctrl-Shift-Del> on the IBM PC.
startup drive	The hard disk or diskette drive that is used to start up the computer.

status byte	The data byte sent by a device when it is serially polled.
status word	Same as `ibsta`. See *ibsta*
System Controller	The single designated Controller that can assert control (become CIC of the GPIB) by sending the Interface Clear (IFC) message. Other devices can become CIC only by having control passed to them.

T

T1	A GPIB timing parameter primarily associated with the data settling time–that is, the time in which new bytes on the DIO lines are allowed to settle before the DAV signal is asserted. T1 ranges from 350 nsec to above 2 µsec.
TAD or talk address	See *MTA*.
Talker	A GPIB device that sends data messages to Listeners.
TCT or take control	The GPIB command used to pass control of the bus from the current Controller to an addressed Talker.
timeout	A feature of the NI-488.2 driver that prevents I/O functions from hanging indefinitely when there is a problem on the GPIB.
TLC	An integrated circuit that implements most of the GPIB Talker, Listener, and Controller functions in hardware.

U

`ud`	A variable name and first argument of each function call that contains the unit descriptor of the GPIB interface board or other GPIB device that is the object of the function. See *unit descriptor.*
unit descriptor	A number that is used by the driver to temporarily identify a device or board that has been opened with the `ibfind` function. The descriptor is not related to the GPIB address of the unit.
UNL or unlisten	The GPIB command used to unaddress any active Listeners.
UNT or untalk	The GPIB command used to unaddress an active Talker.

W

warm startup	See *startup*.

Index