# NI-488.2™
# User Manual for DOS

**January 1996 Edition**

**Part Number 370904A-01**

**National Instruments Corporate Headquarters**
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 794-0100
Technical support fax: (800) 328-2203
(512) 794-5678

**Branch Offices:**
Australia 03 9 879 9422, Austria 0662 45 79 90 0, Belgium 02 757 00 20,
Canada (Ontario) 519 622 9310, Canada (Québec) 514 694 8521,
Denmark 45 76 26 00, Finland 90 527 2321, France 1 48 14 24 24,
Germany 089 741 31 30, Hong Kong 2645 3186, Italy 02 48301892,
Japan 03 5472 2970, Korea 02 596 7456, Mexico 95 800 010 0793,
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

# Limited Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

# Copyright



# Trademarks



# WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

# Contents

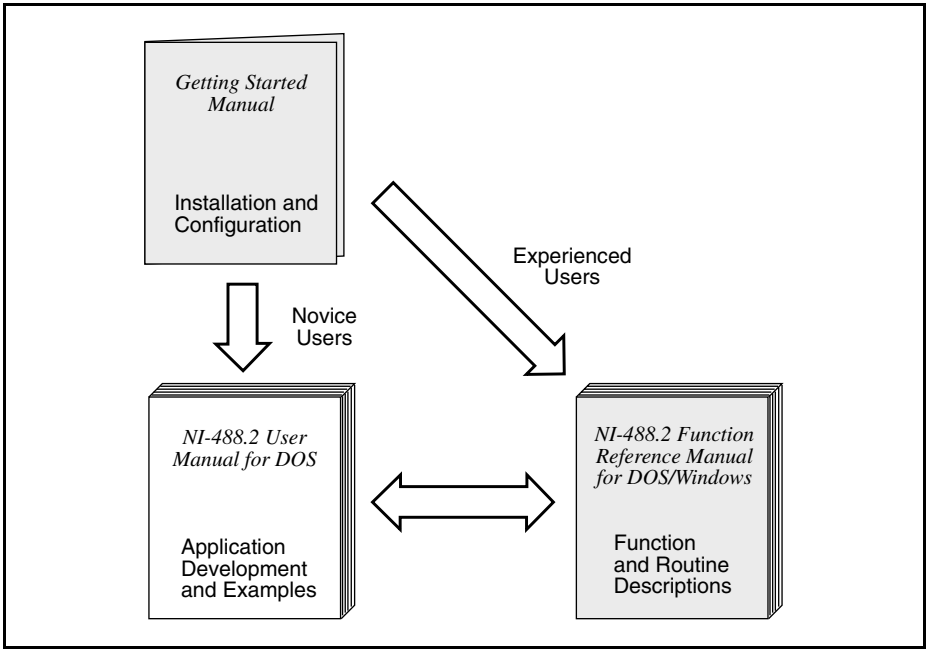
## Figures

## Tables

# About This Manual

This manual describes the features and functions of the NI-488.2 software for DOS. The NI-488.2 software is meant to be used with Microsoft MS-DOS (version 3.0 or higher) or equivalent. This manual assumes that you are already familiar with DOS.

## How to Use This Manual Set




Use the getting started manual to install and configure your GPIB hardware and NI-488.2 software for DOS.

Use the *NI-488.2 User Manual for DOS* to learn the basics of GPIB and how to develop an application program. The user manual also contains debugging information and detailed examples.

Use the *NI-488.2 Function Reference Manual for DOS/Windows* for specific information about each NI-488 function and NI-488.2 routine such as format, parameters, and possible errors.

If you ordered a kit from National Instruments that includes the GPIB analyzer software, you also received documentation for the GPIB analyzer. You can only use the GPIB analyzer in Windows.

## Organization of This Manual

This manual is organized as follows:

- Chapter 1, *Introduction*, gives an overview of GPIB and the NI-488.2 software.
- Chapter 2, *Application Examples*, contains nine sample applications designed to illustrate specific GPIB concepts and techniques that can help you write your own applications. The description of each example includes the programmer's task, a program flowchart, and numbered steps which correspond to the numbered blocks on the flowchart.
- Chapter 3, *Developing Your Application,* explains how to develop a GPIB application program using NI-488 functions and NI-488.2 routines.
- Chapter 4, *Debugging Your Application,* describes several ways to debug your application program.
- Chapter 5, *ibic–Interface Bus Interactive Control*, introduces you to `ibic`, the interactive control program that you can use to communicate with GPIB devices interactively.
- Chapter 6, *appmon–GPIB Applications Monitor*, explains how to install, configure, and use `appmon`, the GPIB applications monitor, which is a useful debugging tool.
- Chapter 7, *GPIB Programming Techniques*, describes techniques for using some NI-488 functions and NI-488.2 routines in your application program.
- Chapter 8, *ibconf–Interface Bus Configuration Utility*, contains a description of `ibconf`, the software configuration program you can use to configure the NI-488.2 software.
- Appendix A, *Status Word Conditions*, gives a detailed description of the conditions reported in the status word, `ibsta`.
- Appendix B, *Error Codes and Solutions*, lists a description of each error, some conditions under which it might occur, and possible solutions.
- Appendix C, *Universal Language Interface*, describes how to install and use the Universal Language Interface (ULI).
- Appendix D, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.

- The *Index* contains an alphabetical list of key terms and topics in this manual, including the page where you can find each one.

## Conventions Used in This Manual

The following conventions are used in this manual.

| | |
|---|---|
| **bold** | Bold text denotes a menu or menu item. |
| *italic* | Italic text denotes emphasis, cross references, field names, or an introduction to a key concept. |
| ***bold italic*** | Bold italic text denotes a note, caution, or warning. |
| `monospace` | Text in this font denotes text or characters that you enter from the keyboard. Sections of code, programming examples, and syntax examples also appear in this font. This font is also used for the proper name of disk drives, paths, directories, device names, variables, and for statements taken from program code. |
| **`bold monospace`** | Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. |
| *`italic monospace`* | Italic 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-C>. |
| IEEE 488 and IEEE 488.2 | *IEEE 488* and *IEEE 488.2* are used throughout this manual to refer to the ANSI/IEEE Standard 488.1-1987 and the ANSI/IEEE Standard 488.2-1992, respectively, which define the GPIB. |
| NI-488.2 software | The term *NI-488.2 software* is used throughout this manual to refer to the NI-488.2 software for DOS unless otherwise noted. |

Abbreviations, acronyms, metric prefixes, mnemonics, symbols, and terms are listed in the *Glossary.*

## 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 488.2-1992, *IEEE Standard Codes, Formats, Protocols, and Common Commands*
- *Microsoft MS-DOS User's Guide*

## 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 D, *Customer Communication*, at the end of this manual.

# Chapter 1
# Introduction

This chapter gives an overview of GPIB and the NI-488.2 software.

## GPIB Overview

The ANSI/IEEE Standard 488.1-1987, also known as GPIB (General Purpose Interface Bus), describes a standard interface for communication between instruments and controllers from various vendors. It contains information about electrical, mechanical, and functional specifications. The GPIB is a digital, 8-bit parallel communications interface with data transfer rates of 1 Mbytes/s and above. The bus supports one System Controller, usually a computer, and up to 14 additional instruments. The ANSI/IEEE Standard 488.2-1987 extends IEEE 488.1 by defining a bus communication protocol, a common set of data codes and formats, and a generic set of common device commands.

### Talkers, Listeners, and Controllers

GPIB devices can be Talkers, Listeners, or Controllers. A Talker sends out data messages. Listeners receive data messages. The Controller, usually a computer, manages the flow of information on the bus. It defines the communication links and sends GPIB commands to devices.

Some devices are capable of playing more than one role. A digital voltmeter, for example, can be a Talker and a Listener. If your personal computer has a National Instruments GPIB interface board and the NI-488.2 software installed, it can function as a Talker, Listener, and Controller.

### Controller-In-Charge and System Controller

You can have multiple Controllers on the GPIB, but only one Controller at a time can be the active Controller, or Controller-In-Charge (CIC). The CIC can either be active or inactive (Standby) Controller. Control can pass from the current CIC to an idle Controller, but only the System Controller, usually a GPIB interface board, can make itself the CIC.

## GPIB Addressing

All devices and boards connected to the GPIB must be assigned a unique GPIB address. The Controller uses the addresses to identify each device when sending or receiving data. A GPIB address is made up of two parts: a primary address and an optional secondary address. You usually set the address using switches on the board or device.

The primary address is a number in the range 0 to 30. The GPIB Controller uses the primary address to form a talk or listen address that is sent over the GPIB when communicating with a device.

A talk address is formed by setting bit 6, the TA (Talk Active) bit of the GPIB address. A listen address is formed by setting bit 5, the LA (Listen Active) bit of the GPIB address. For example, if a device is at address 1, the Controller sends hex 41 (address 1 with bit 6 set) to make the device a Talker. Because the Controller is usually at primary address 0, it sends hex 20 (address 0 with bit 5 set) to make itself a Listener. Figure 1-1 shows the configuration of the GPIB address bits.

| **Bit Position** | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **Meaning** | 0 | TA | LA | GPIB Primary Address (range 0 to 30) | | | | |

Figure 1-1. GPIB Address Bits

With some devices, you can use secondary addressing. A secondary address is a number in the range hex 60 to hex 7E. When secondary addressing is in use, the Controller sends the primary talk or listen address of the device followed by the secondary address of the device.

## Sending Messages Across the GPIB

Devices on the bus communicate by sending messages. Signals and lines transfer these messages across the GPIB interface, which consists of 16 signal lines and eight ground return (shield drain) lines. The 16 signal lines are discussed in the following sections.

### Data Lines

Eight data lines, DIO1 through DIO8, carry both data and command messages.

### Handshake Lines

Three hardware handshake lines asynchronously control the transfer of message bytes between devices. This process is a three-wire interlocked handshake, and it guarantees that devices send and receive message bytes on the data lines without transmission error. Table 1-1 summarizes the GPIB handshake lines.

Table 1-1. GPIB Handshake Lines

| Line | Description |
|---|---|
| NRFD (not ready for data) | Listening device is ready/not ready to receive a message byte. Also used by the Talker to signal high-speed transfers (HS488). |
| NDAC (not data accepted) | Listening device has/has not accepted a message byte. |
| DAV (data valid) | Talking device indicates signals on data lines are stable (valid) data. |

### Interface Management Lines

Five GPIB hardware lines manage the flow of information across the bus. Table 1-2 summarizes the GPIB interface management lines.

Table 1-2. GPIB Interface Management Lines

| Line | Description |
|---|---|
| ATN (attention) | Controller drives ATN true when it sends commands and false when it sends data messages. |
| IFC (interface clear) | System Controller drives the IFC line to initialize the bus and make itself CIC. |
| REN (remote enable) | System Controller drives the REN line 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. |
| EOI (end or identify) | Talker uses the EOI line to mark the end of a data message. Controller uses the EOI line when it conducts a parallel poll. |

## Setting Up and Configuring Your System

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. With this design, you can link devices in a linear configuration, a star configuration, or a combination of the two. Figure 1-2 shows the linear and star configurations.

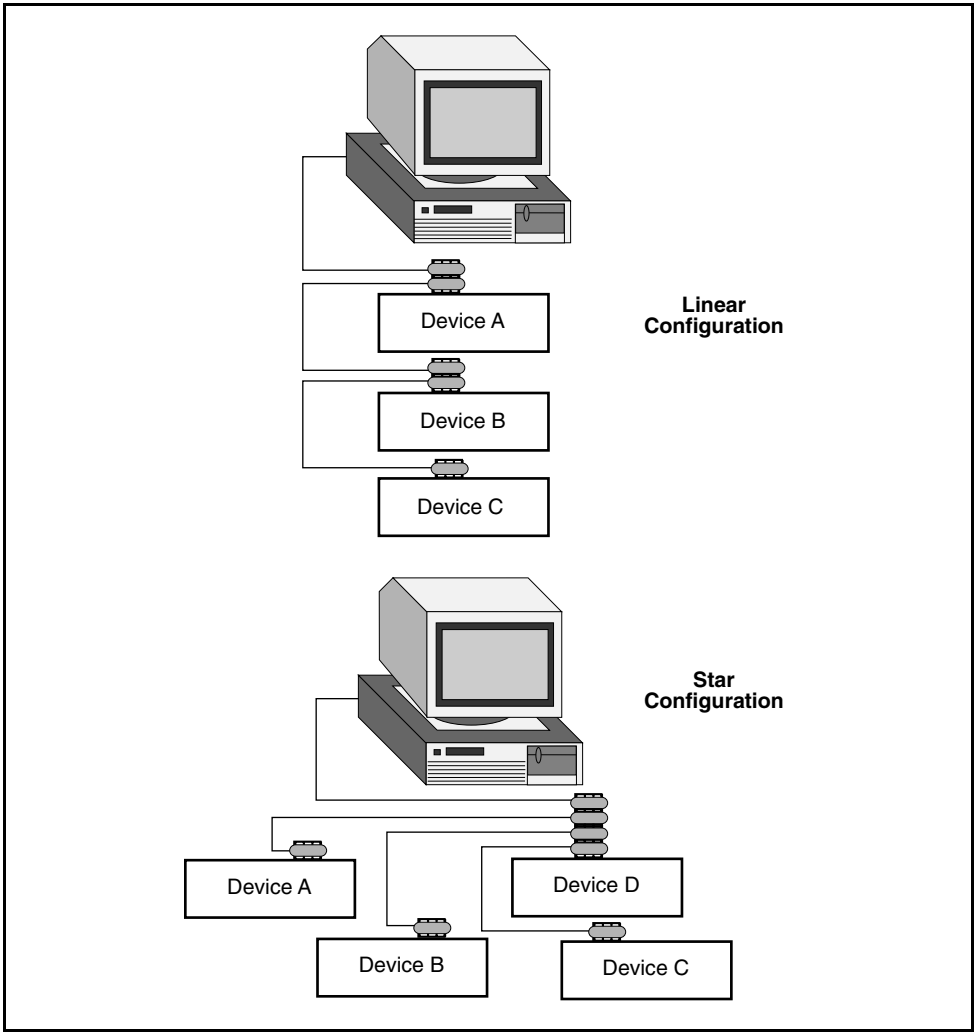



Figure 1-2. Linear and Star System Configuration

**Controlling More Than One Board**

Multiboard drivers, such as the NI-488.2 driver for DOS, can control more than one interface board. Figure 1-3 shows an example of a multiboard system configuration. `gpib0` is the access board for the voltmeter, and `gpib1` is the access board for the plotter and printer. The control functions of the devices automatically access their respective boards.

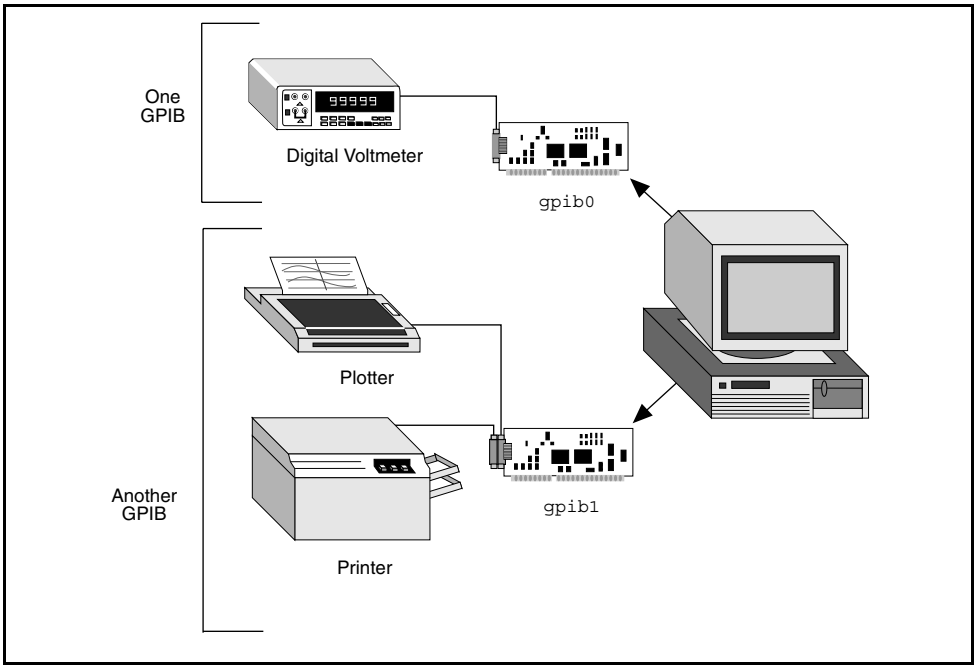



Figure 1-3. Example of Multiboard System Setup

**Configuration Requirements**

To achieve the high data transfer rate that the GPIB was designed for, you must limit the physical distance between devices and the number of devices on the bus. 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 m.
- A maximum of 15 devices connected to each bus, with at least two-thirds powered on.

For high-speed operation, the following restrictions apply:

- All devices in the system must be powered on.
- Cable lengths should be as short as possible up to a maximum of 15 m of cable in each system.
- At least one equivalent device load per meter of cable.

If you want to exceed these limitations, you can use bus extenders to increase the cable length or expanders to increase the number of device loads. Extenders and expanders are available from National Instruments.

# The NI-488.2 Software Components

The following section highlights important elements of the NI-488.2 software for DOS and describes the function of each element.

## NI-488.2 Driver and Driver Utilities

The NI-488.2 software includes the following driver and utility files:

- `readme.txt` is a documentation file that contains important information about the NI-488.2 software and a description of any new features. Before you use the software, read this file for the most recent information.
- `install.exe` is a menu-driven program that installs the NI-488.2 software.
- `gpib.com` is the NI-488.2 driver that DOS loads at system startup.
- `ibdiag.exe` is a program that you can use to test the GPIB hardware and ensure that it is functioning properly.
- `ibtest.exe` is the NI-488.2 software installation test.
- `gpibinfo.exe` is a utility you can use to obtain information about your GPIB hardware and software, such as the version of the NI-488.2 software and the type of interface board you are using.
- `ibic.exe` is an interactive control program that you use to communicate with the GPIB devices interactively using NI-488.2 functions and routines. It helps you to learn the NI-488.2 routines and to program your instrument or other GPIB devices.
- `ibconf.exe` is a software configuration program that changes the configuration parameters of the NI-488.2 software.

- `appmon.exe` is the GPIB applications monitor program. It is a debugging tool that you can use to monitor the NI-488.2 calls made by your DOS applications.

- `ibtrap.exe` is a program you can use to configure the applications monitor.

## C Language Files

The NI-488.2 software includes the following C language files:

- `readme.mc` is a documentation file that contains information about the C language interface.

- `mcib.lib` is the Microsoft C (version 5.1 or higher) and Borland C++ (version 2.0 or higher) language interface library. You must link this library with your application program so that your program can access the NI-488.2 driver. This file does not support the Borland C++ huge memory model. Contact National Instruments for the language interface for the Borland C++ huge memory model.

- `decl.h` is an include file. It contains NI-488 function and NI-488.2 routine prototypes and various predefined constants.

## BASIC Language Files

The NI-488.2 software includes language interface files for Microsoft Professional BASIC (version 7.0 or higher), Microsoft Visual Basic for DOS (version 1.0 or higher), QuickBASIC (version 4.0 or higher), BASICA, and GWBASIC. The files are as follows:

- `readme.mb` is a documentation file that contains information about the Microsoft BASIC and Visual Basic language interface.

- `mbib.obj` is a binary language interface file that gives an application program written in Microsoft BASIC (version 7.0 or higher) and Microsoft Visual Basic (version 1.0 or higher) access to the NI-488.2 driver.

- `mbdecl.bas` is a declaration file that contains code you should place at the beginning of Microsoft BASIC and Visual Basic application programs.

- `readme.qb` is a documentation file that contains information about the QuickBASIC language interface.

- `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 driver.

- `qbdecl.bas` is a declaration file that contains code you should place at the beginning of QuickBASIC application programs.

- `readme.ba` is a documentation file that contains information about the BASICA/GWBASIC language interface.
- `bib.m` is a binary language interface file that gives a BASICA/GWBASIC program access to the NI-488.2 driver.
- `decl.bas` is a declaration file that contains code you should place at the beginning of BASICA/GWBASIC application programs.

## Universal Language Interface File

The NI-488.2 software contains the universal language interface (ULI) file `uli.com`. Use the universal language interface only if you have an existing application that uses HP-style calls.

## Sample Application Files

The NI-488.2 software includes nine sample applications along with source code for C, Microsoft BASIC, QuickBASIC, and BASICA. For a detailed description of the sample application files, refer to Chapter 2, *Application Examples*.

## How the NI-488.2 Software Works with DOS

The NI-488.2 driver operates as a standard DOS device driver, which is loaded at system startup.

Figure 1-4 shows how the NI-488.2 software works with DOS and your GPIB hardware.

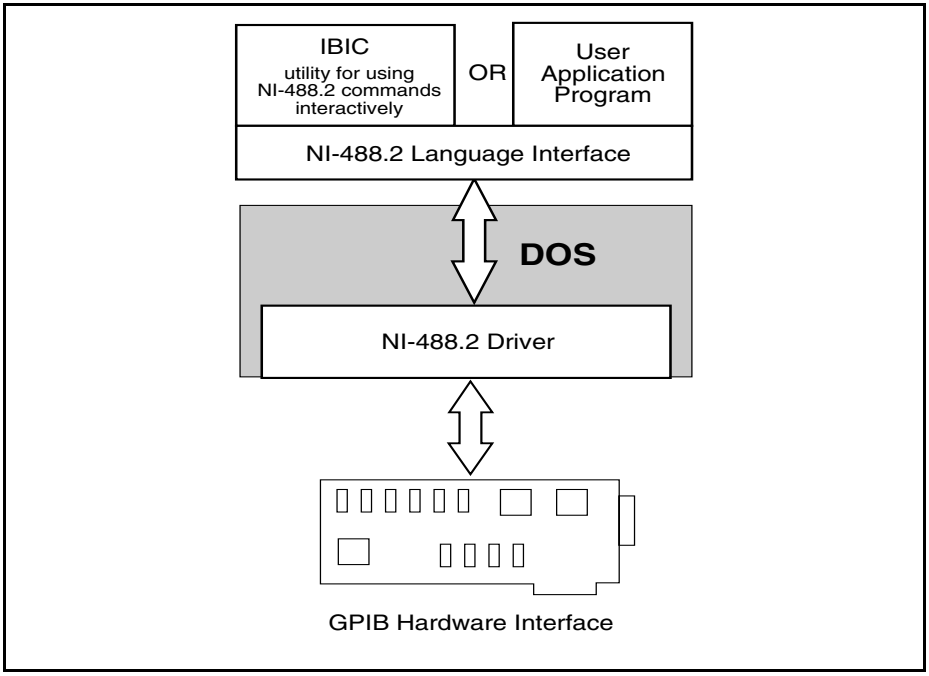



Figure 1-4. How the NI-488.2 Software Works with DOS

## Unloading and Reloading the NI-488.2 Driver

The NI-488.2 driver is loaded by a command in the `config.sys` file when DOS is booted. If you ever decide to unload the NI-488.2 driver, change this command to a remark by adding `rem`. After you change the command to a remark, the NI-488.2 driver is not loaded the next time you reboot your system. To prevent the NI-488.2 driver from being loaded, edit the `config.sys` file as follows:

1. Locate the `config.sys` file in the root directory of your boot drive (usually `c:`).
2. Locate the command line that is of the following form:

   `device = drive:\path\gpib.com`

where *drive* is the drive (usually c) and *path* is the path where the NI-488.2 software was installed (for example, c:\at-gpib).

3. Change the command line to a remark by adding rem as follows:

   ```
   rem device = drive:\path\gpib.com
   ```

4. Reboot your computer so that the change can take effect.

If you later decide to use the NI-488.2 driver again, remove rem from the line and reboot the system.

# Chapter 2
# Application Examples

This chapter contains nine sample applications designed to illustrate specific GPIB concepts and techniques that can help you write your own applications. The description of each example includes the programmer's task, a program flowchart, and numbered steps which correspond to the numbered blocks on the flowchart.

Use this chapter along with your NI-488.2 software, which includes the C and BASIC source code for each of the nine examples. The programs are listed in order of increasing complexity. If you are new to GPIB programming, you might want to study the contents and concepts of the first sample, `simple.c`, before moving on to more complex examples.

The following example programs are included with your NI-488.2 software:

- `simple.c` is the source code file for Example 1. It illustrates how you can establish communication between a host computer and a GPIB device.
- `clr_trg.c` is the source code file for Example 2. It illustrates how you can clear and trigger GPIB devices.
- `asynch.c` is the source code file for Example 3. It illustrates how you can perform non-GPIB tasks while data is being transferred over the GPIB.
- `eos.c` is the source code file for Example 4. It illustrates the concept of the end-of-string (EOS) character.
- `rqs.c` is the source code file for Example 5. It illustrates how you can communicate with GPIB devices that use the GPIB SRQ line to request service. This sample is written using NI-488 functions.
- `easy4882.c` is the source code file for Example 6. It is an introduction to NI-488.2 routines.
- `rqs4882.c` is the source code file for Example 7. It uses NI-488.2 routines to communicate with GPIB devices that use the GPIB SRQ line to request service.
- `ppoll.c` is the source code file for Example 8. It uses NI-488.2 routines to conduct parallel polls.
- `non_cic.c` is the source code file for Example 9. It illustrates how you can use the NI-488.2 driver in a non-Controller application.

# Example 1: Basic Communication

This example focuses on the basics of establishing communication between a host computer and a GPIB device.

A technician needs to monitor voltage readings using a GPIB multimeter. His computer is equipped with an IEEE 488.2 interface board. The NI-488.2 software is installed and a GPIB cable runs from the computer to the GPIB port on the multimeter.

The technician is familiar with the multimeter remote programming command set. This list of commands is specific to his multimeter and is available from the multimeter manufacturer.

He sets up the computer to direct the multimeter to take measurements and record each measurement as it occurs. To do this, he has written an application that uses some simple high-level GPIB commands. The following steps correspond to the program flowchart in Figure 2-1.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application sends the multimeter an instruction, setting it up to take voltage measurements in autorange mode.

3. The application sends the multimeter an instruction to take a voltage measurement.

4. The application tells the multimeter to transmit the data it has acquired to the computer.

   The process of requesting a measurement and reading from the multimeter (Steps 3 and 4) is repeated as long as there are readings to be obtained.

5. As a cleanup step before exiting, the application returns the interface board to its original state by taking it offline.

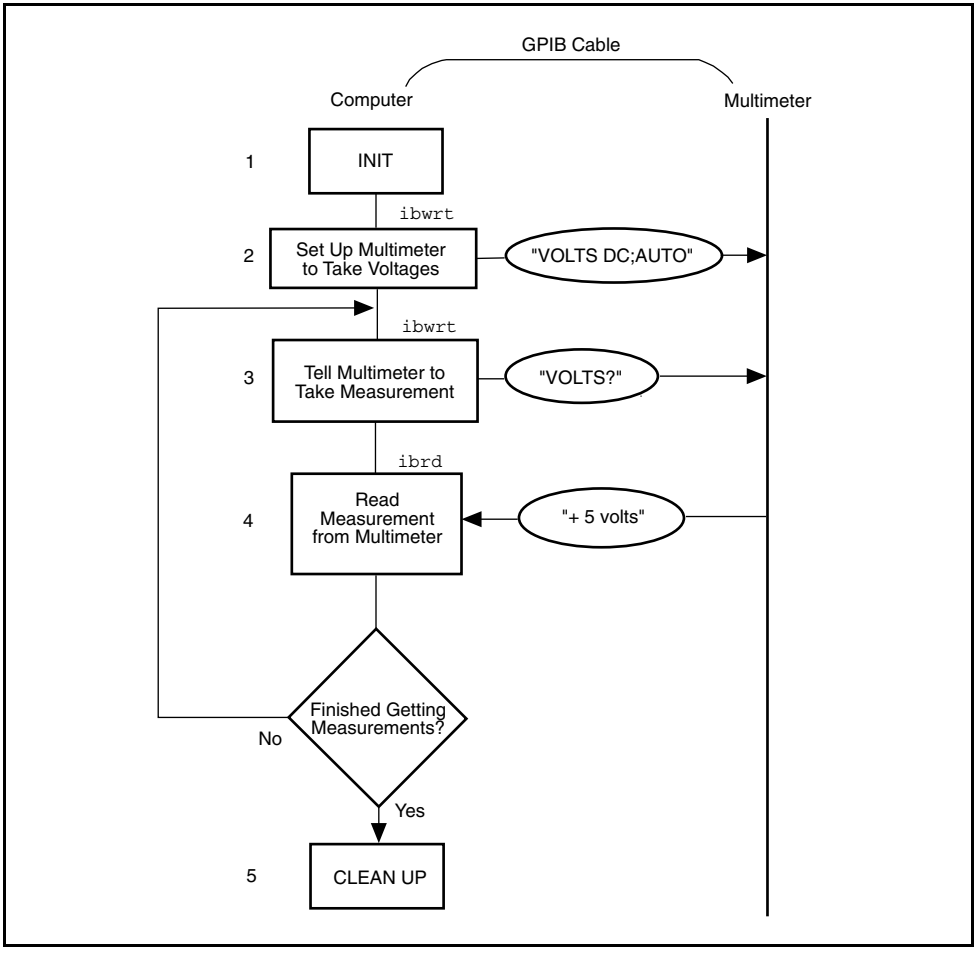



Figure 2-1. Program Flowchart for Example 1

## Example 2: Clearing and Triggering Devices

This example illustrates how you can clear and trigger GPIB devices.

Two freshman physics lab partners are learning how to use a GPIB digital oscilloscope. They have successfully loaded the NI-488.2 software on a personal computer and connected their GPIB board to a GPIB digital oscilloscope. Their current lab assignment is to write a small application to practice using the oscilloscope and its command set using high-level GPIB commands. The following steps correspond to the program flowchart in Figure 2-2.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application sends a GPIB clear command to the oscilloscope. This command clears the internal registers of the oscilloscope, reinitializing it to default values and settings.

3. The application sends a command to the oscilloscope telling it to read a waveform each time it is triggered. Predefining the task in this way decreases the execution time required. Each trigger of the oscilloscope is now sufficient to get a new run.

4. The application sends a GPIB trigger command to the oscilloscope which causes it to acquire data.

5. The application queries the oscilloscope for the acquired data. The oscilloscope sends the data.

6. The application reads the data from the oscilloscope.

7. The application calls an external graphics routine to display the acquired waveform.

   Steps 4, 5, 6, and 7 are repeated until all of the desired data has been acquired by the oscilloscope and received by the computer.

8. As a cleanup step before exiting, the application returns the interface board to its original state by taking it offline.

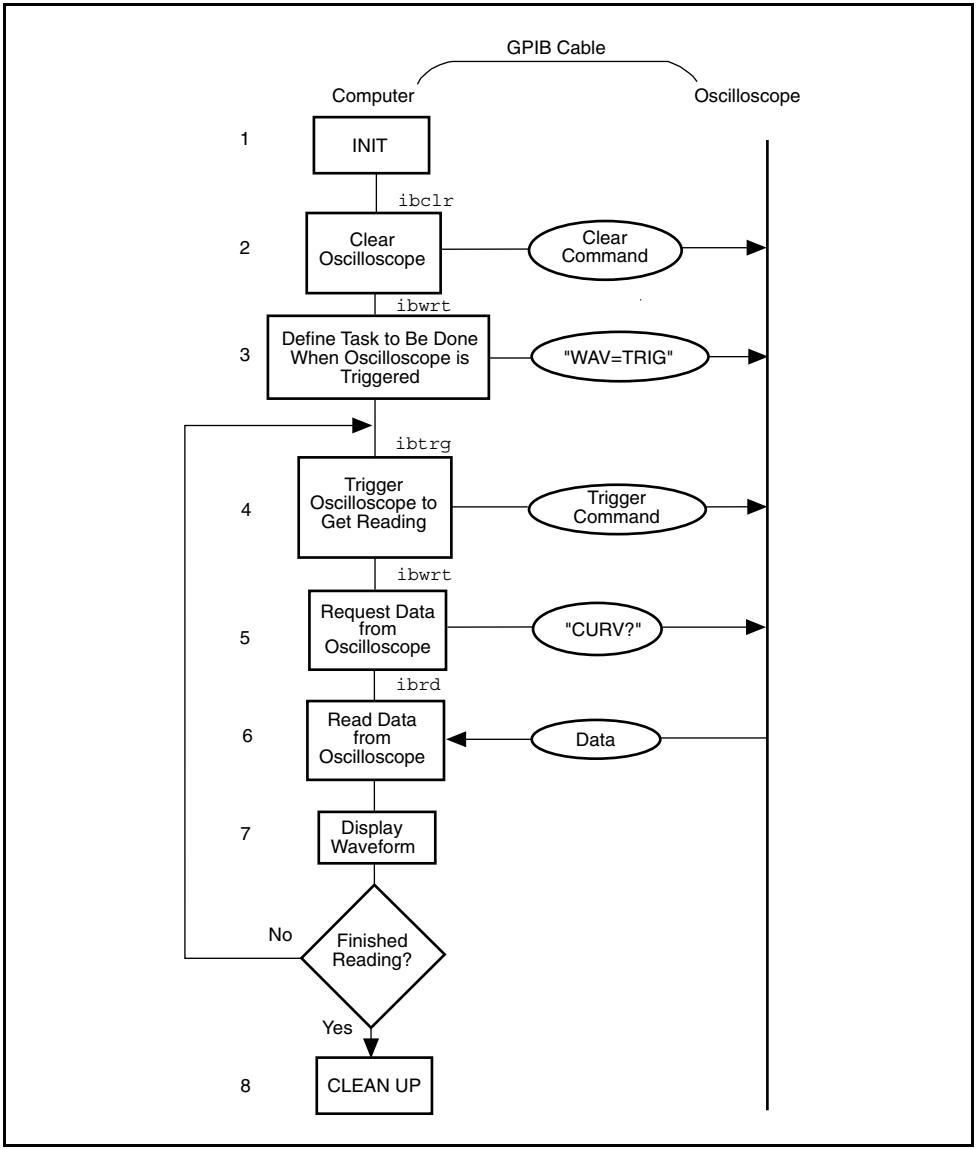



Figure 2-2. Program Flowchart for Example 2

# Example 3: Asynchronous I/O

This example illustrates how an application conducts data transfers with a GPIB device and immediately returns to perform other non-GPIB related tasks while GPIB I/O is occurring in the background. This asynchronous mode of operation is particularly useful when the requested GPIB activity may take some time to complete.

In this example, a research biologist is trying to obtain accurate CAT scans of a laboratory animal's liver. She will print out a color copy of each scan as it is acquired. The entire operation is computer-controlled. The CAT scan machine sends the images it acquires to a computer that is connected to a GPIB color printer and has the NI-488.2 software installed. The biologist is familiar with the command set of her color printer, as described in the color printer's user manual. She acquires and prints images with the aid of an application program she wrote using high-level GPIB commands. The following steps correspond to the program flowchart in Figure 2-3.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. An image is scanned in.

3. The application sends the GPIB printer a command to print the new image and immediately returns without waiting for the I/O operation to be completed.

4. The application saves the image obtained to a file.

5. The application inquires as to whether the printing operation has completed by issuing a GPIB wait command. If the status reported by the wait command indicates completion (CMPL is in the status returned) and more scans need to be acquired, Steps 2 through 5 are repeated until the scans have all been acquired. If the status reported by the wait command in Step 5 does not indicate that printing is finished, statistical computations are performed on the scan obtained and Step 5 is repeated.

6. As a cleanup step before exiting, the application returns the interface board to its original state by taking it offline.

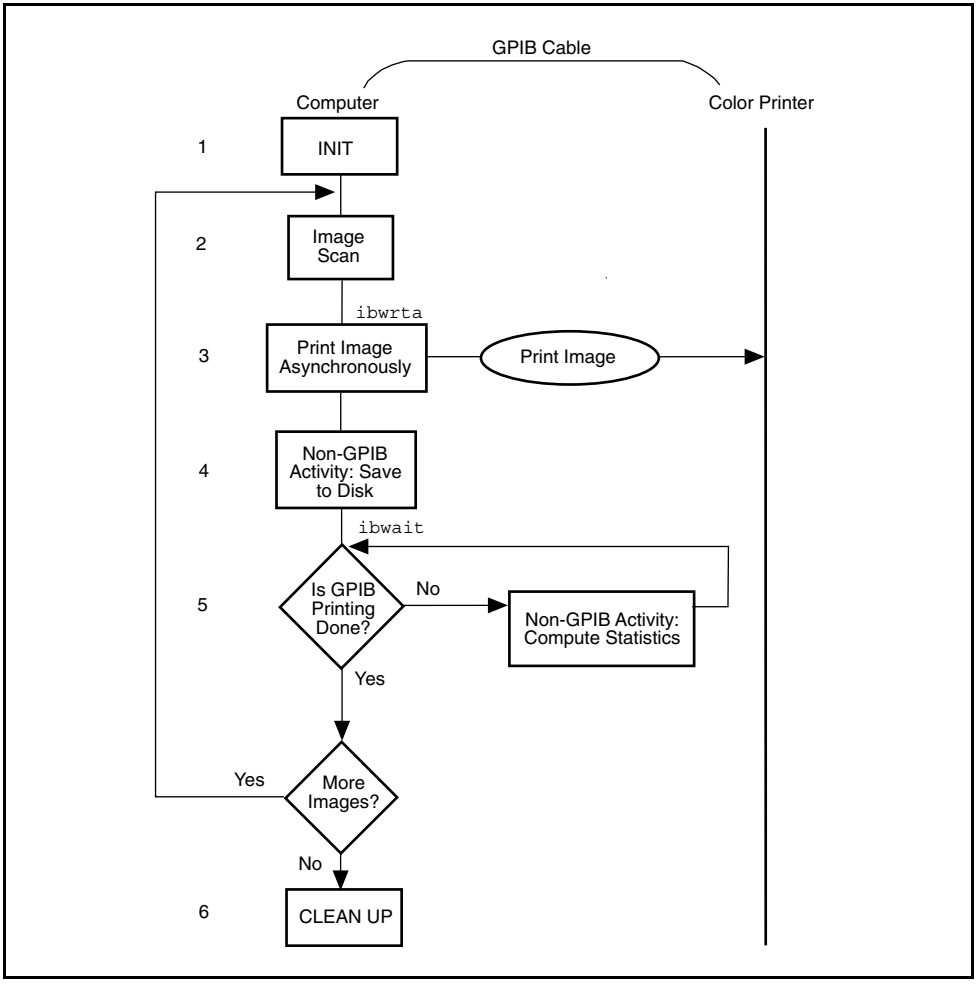



Figure 2-3. Program Flowchart for Example 3

## Example 4: End-of-String Mode

This example illustrates how to use the end-of-string modes to detect that the GPIB device has finished sending data.

A journalist is using a GPIB scanner to scan some pictures into his personal computer for a news story. A GPIB cable runs between the scanner and the computer. He is using an application written by an intern in the department who has read the scanner's instruction manual and is familiar with the scanner's programming requirements. The following steps correspond to the program flowchart in Figure 2-4.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application sends a GPIB clear message to the scanner, initializing it to its power-on defaults.

3. The scanner needs to detect a delimiter indicating the end of a command. In this case, the scanner expects the commands to be terminated with <CR><LF> (carriage return, `\r`, and linefeed, `\n`). The application sets its end-of-string (EOS) byte to <LF>. The linefeed code indicates to the scanner that no more data is coming, and is called the end-of-string byte. It flags an end-of-string condition for this particular GPIB scanner. The same effect could be accomplished by asserting the EOI line when the command is sent.

4. With the exception of the scan resolution, all the default settings are appropriate for the task at hand. The application changes the scan resolution by writing the appropriate command to the scanner.

5. The scanner sends back information describing the status of the *change resolution* command. This is a string of bytes terminated by the end-of-string character to tell the application it is done changing the resolution.

6. The application starts the scan by writing the scan command to the scanner.

7. The application reads the scan data into the computer.

8. As a cleanup step before exiting, the application returns the interface board to its original state by taking it offline.

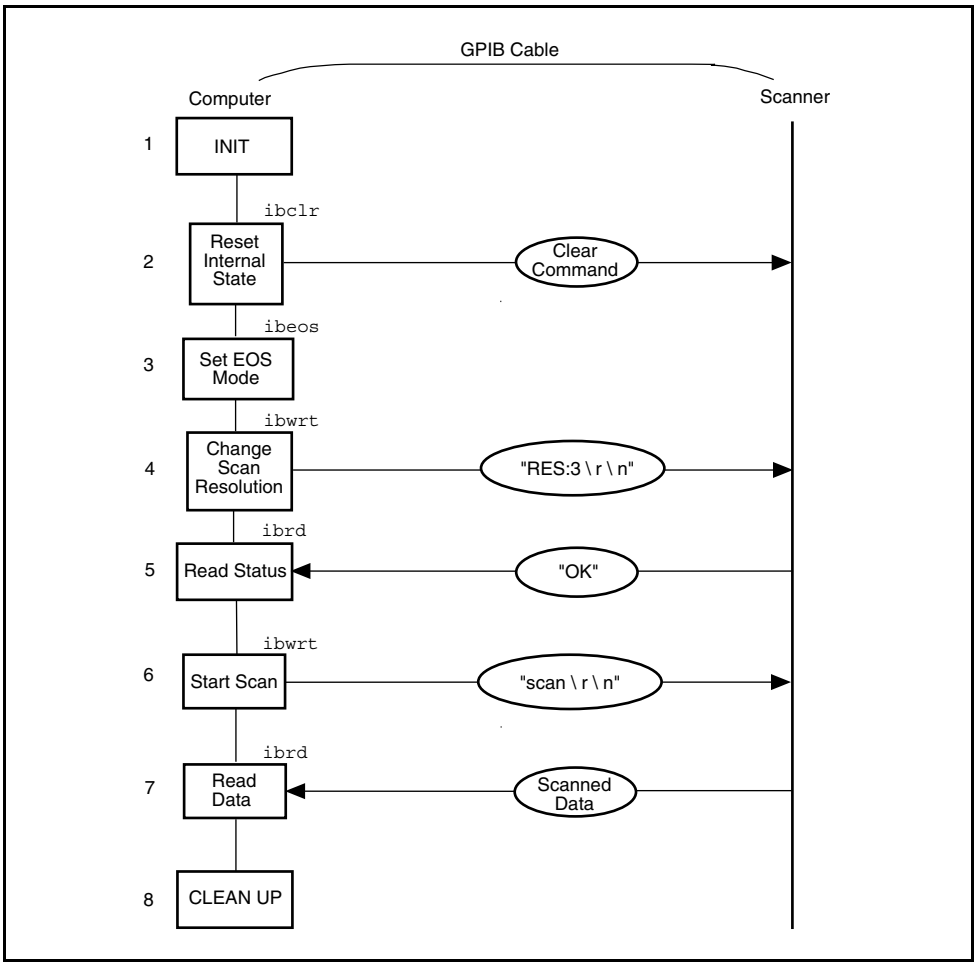



Figure 2-4. Program Flowchart for Example 4

# Example 5: Service Requests

This example illustrates how an application communicates with a GPIB device that uses the GPIB service request (SRQ) line to indicate that it needs attention.

A graphic arts designer is transferring digital images stored on her computer to a roll of color film, using a GPIB digital film recorder. A GPIB cable connects the GPIB port on the film recorder to the IEEE 488.2 interface board installed in her computer. She has installed the NI-488.2 software on the host computer and is familiar with the programming instructions for the film recorder, as described in the film recorder's user manual. She places a fresh roll of film in the camera and launches a simple application she has written using high-level GPIB commands. With the aid of the application, she records a few images on film. The following steps correspond to the program flowchart in Figure 2-5.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application brings the film recorder to a ready state by issuing a device clear instruction. The film recorder is now set up for operation using its default values. (The graphic arts designer has previously established that the default values for the film recorder are appropriate for the type of film she is using).

3. The application advances the new roll of film into position so the first image can be exposed on the first frame of film. This is done by sending the appropriate instructions as described in the film recorder programming guide.

4. The application waits for the film recorder to signify that it is done loading the film, by waiting for RQS (request for service). The film recorder asserts the GPIB SRQ line when it has finished loading the film.

5. As soon as the film recorder asserts the GPIB SRQ line, the application's wait for the RQS event completes. The application conducts a serial poll by sending a special command message to the film recorder that directs it to return a response in the form of a serial poll status byte. This byte contains information indicating what kind of service the film recorder is requesting or what condition it is flagging. In this example, it indicates the completion of a command.

6. A color image transfers to the digital film recorder in three consecutive passes–one pass each for the red, green and blue components of the image. Sub-steps 6a, 6b, and 6c are repeated for each of the passes:

    6a. The application sends a command to the film recorder directing it to accept data to create a single pass image. The film recorder asserts the SRQ line as soon as a pass is completed.

    6b. The application waits for RQS.

6c. When the SRQ line is asserted, the application serial polls the film recorder to see if it requested service, as in Step 5.

7. The application issues a command to the film recorder to advance the film by one frame. The advance occurs successfully unless the end of film is reached.

8. The application waits for RQS, which completes when the film recorder asserts the SRQ line to signal it is done advancing the film.

9. As soon as the application's wait for RQS completes, the application serial polls the film recorder to see if it requested service, as in Step 5. The returned serial poll status byte indicates either of two conditions–the film recorder finished advancing the film as requested or the end of film was reached and it can no longer advance. Steps 6 through 9 are repeated as long as film is in the camera and more images need to be recorded.

10. As a cleanup step before exiting, the application returns the interface board to its original state by taking it offline.

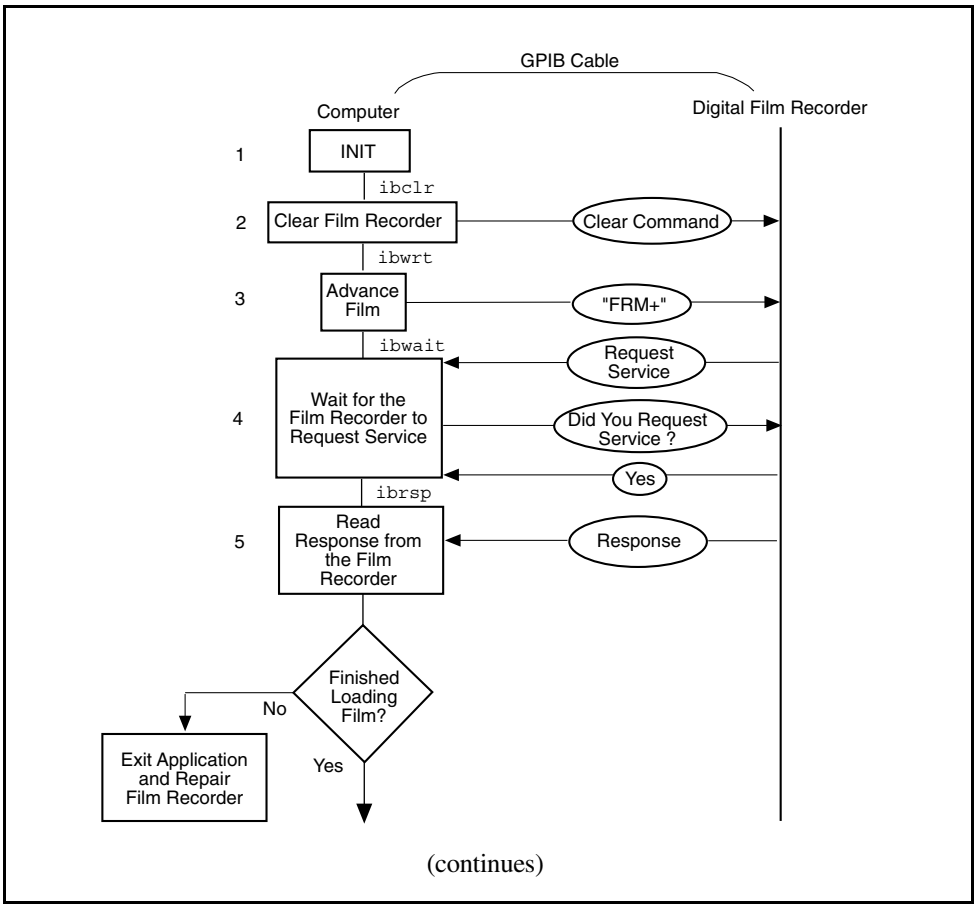



Figure 2-5. Program Flowchart for Example 5

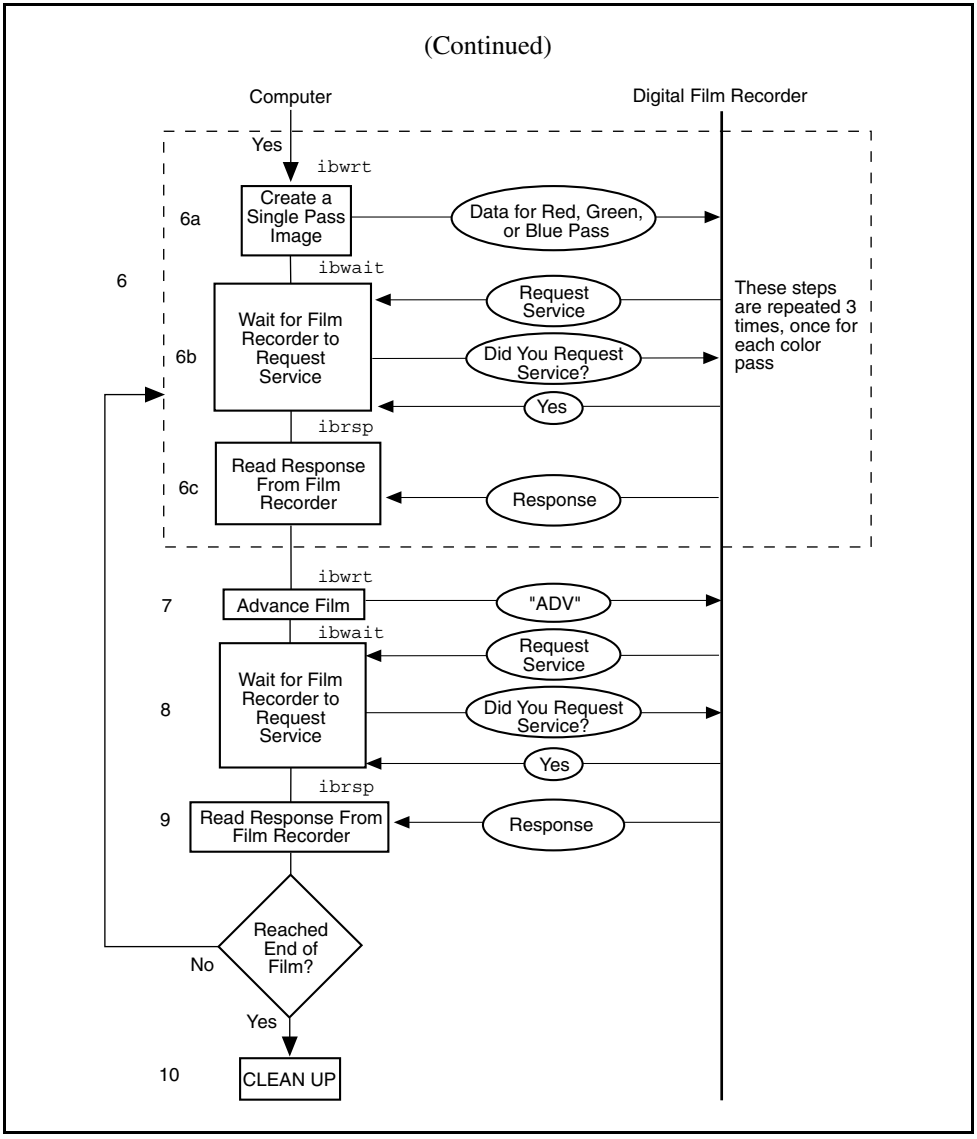



Figure 2-5. Program Flowchart for Example 5 (Continued)

# Example 6: Basic Communication with IEEE 488.2-Compliant Devices

This example is an introduction to communicating with IEEE 488.2-compliant devices.

A test engineer in a metal factory is using IEEE 488.2-compliant tensile testers to determine the strength of metal rods as they come out of production. There are several tensile testers and they are all connected to a central computer equipped with an IEEE 488.2 interface board. These machines are fairly voluminous and it is difficult for the engineer to reach the address switches of each machine. For the purposes of his future work with these tensile testers, he needs to determine the setting of each GPIB address switch. He can do so with the aid of a simple application he has written. The following steps correspond to the program flowchart in Figure 2-6.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application issues a command to detect the presence of listening devices on the GPIB and compiles a list of the addresses of all such devices.

3. The application sends an identification query (`"*IDN?"`) to all of the devices detected on the GPIB in Step 2.

4. The application reads the identification information returned by each of the devices as it responds to the query in Step 3.

5. As a cleanup step before exiting, the application returns the interface board to its original state by taking it offline.

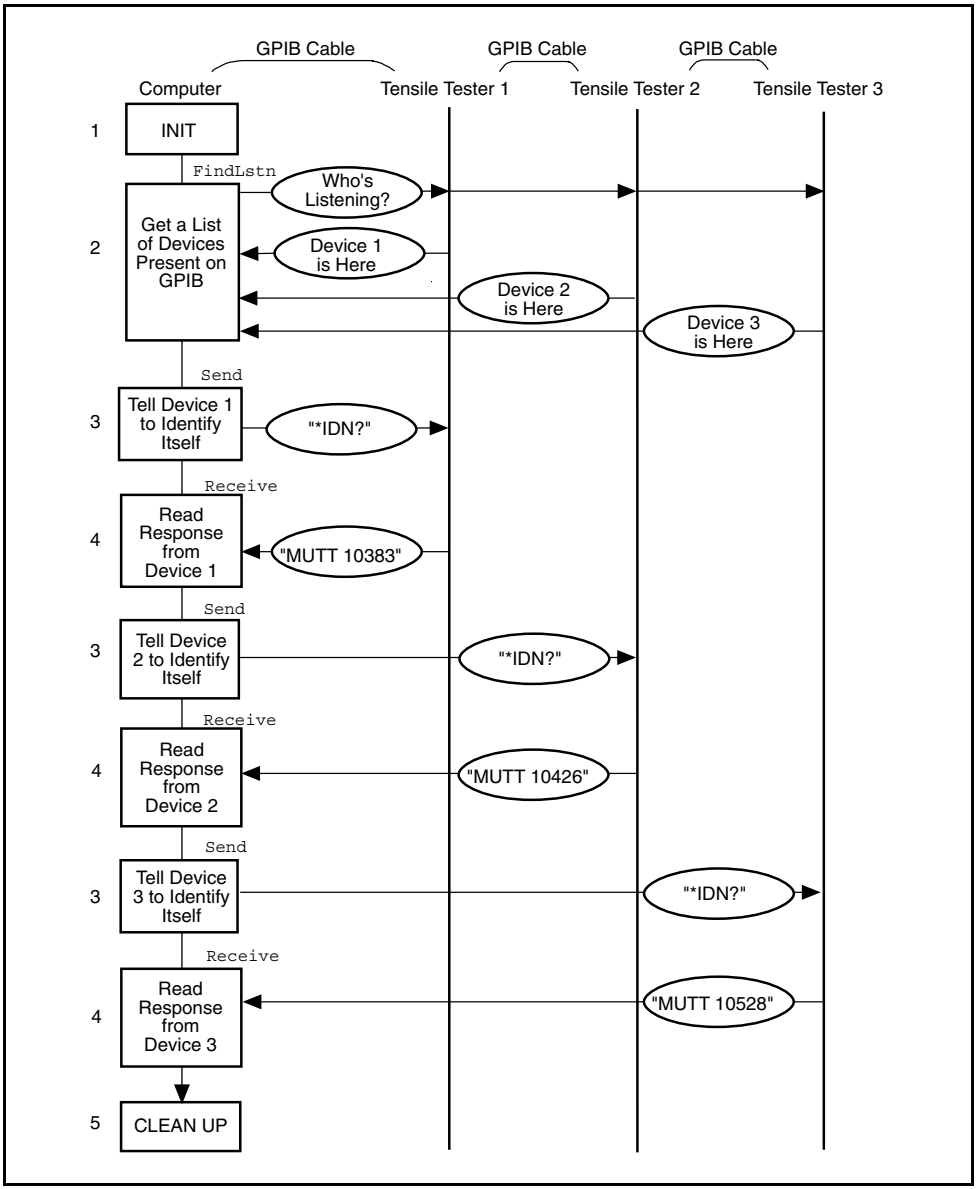



Figure 2-6. Program Flowchart for Example 6

# Example 7: Serial Polls Using NI-488.2 Routines

This example illustrates how you can take advantage of the NI-488.2 routines to reduce the complexity of performing serial polls of multiple devices.

A candy manufacturer is using GPIB strain gauges to measure the consistency of the syrup used to make candy. The plant has four big mixers containing syrup. The syrup has to reach a certain consistency to make good quality candy. This is measured by strain gauges that monitor the amount of pressure used to move the mixer arms. When a certain consistency is reached, the mixture is removed and a new batch of syrup is poured in the mixer. The GPIB strain gauges are connected to a computer with an IEEE 488.2 interface board and the NI-488.2 software installed. The process is controlled by an application that uses NI-488.2 routines to communicate with the IEEE 488.2-compliant strain gauges. The following steps correspond to the program flowchart in Figure 2-7.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application configures the strain gauges to request service when they have a significant pressure reading or a mechanical failure occurs. They signal their request for service by asserting the SRQ line.

3. The application waits for one or more of the strain gauges to indicate that they have a significant pressure reading. This wait event ends as soon as the SRQ line is asserted.

4. The application serial polls each of the strain gauges to see if it requested service.

5. Once the application has determined which one of the strain gauges requires service, it takes a reading from that strain gauge.

6. If the reading matches the desired consistency, a dialog window appears on the computer screen and prompts the mixer operator to remove the mixture and start a new batch. Otherwise, a dialog window prompts the operator to service the mixer in some other way.

   Steps 3 through 6 are repeated as long as the mixers are in operation.

7. After the last batch of syrup has been processed, the application returns the interface board to its original state by taking it offline.

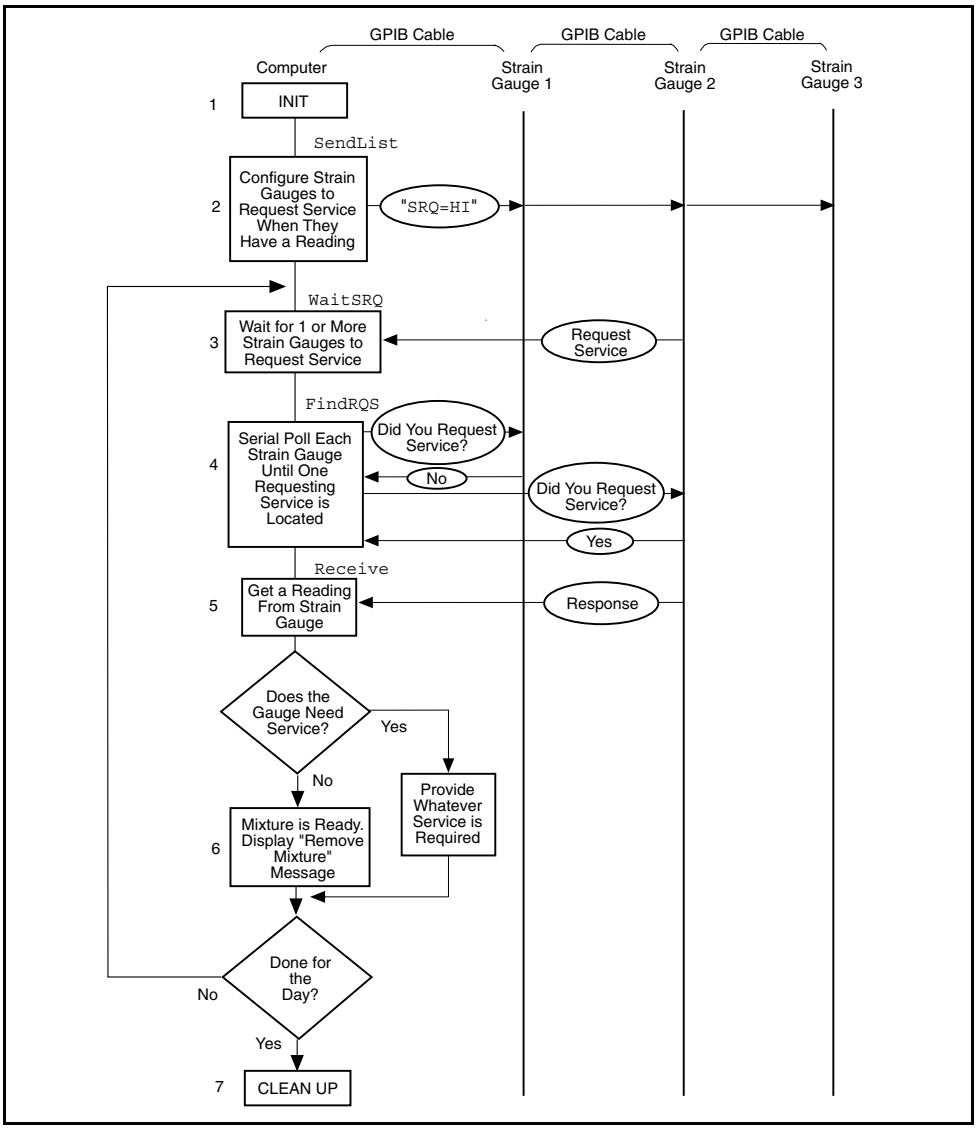



Figure 2-7. Program Flowchart for Example 7

# Example 8: Parallel Polls

This example illustrates how you can use NI-488.2 routines to obtain information from several IEEE 488.2-compliant devices at once using a procedure called parallel polling.

The process of manufacturing a particular alloy involves bringing three different metals to specific temperatures before mixing them to form the alloy. Three vats are used, each containing a different metal. Each is monitored by a GPIB ore monitoring unit. The monitoring unit consists of a GPIB temperature transducer and a GPIB power supply. The temperature transducer is used to probe the temperature of each metal. The power supply is used to start a motor to pour the metal into the mold when it reaches a predefined temperature. The three monitoring units are connected to the IEEE 488.2 interface board of a computer that has the NI-488.2 software installed. An application using NI-488.2 routines operates the three monitoring units. The application will obtain information from the multiple units by conducting a parallel poll, and will then determine when to pour the metals into the mixture tank. The following steps correspond to the program flowchart in Figure 2-8.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application configures the temperature transducer in the first monitoring unit by choosing which of the eight GPIB data lines the transducer uses to respond when a parallel poll is conducted. The application also sets the temperature threshold. The transducer manufacturer has defined the individual status (`ist`) bit to be true when the temperature threshold is reached, and the configured status mode of the transducer is *assert the data line*. When a parallel poll is conducted, the transducer asserts its data line if the temperature has exceeded the threshold.

3. The application configures the temperature transducer in the second monitoring unit for parallel polls.

4. The application configures the temperature transducer in the third monitoring unit for parallel polls.

5. The application conducts non-GPIB activity while the metals are heated.

6. The application conducts a parallel poll of all three temperature transducers to determine whether the metals have reached the appropriate temperature. Each transducer asserts its data line during the configuration step if its temperature threshold has been reached.

7. If the response to the poll indicates that all three metals are at the appropriate temperature, the application sends a command to each of the three power supplies, directing them to power on. Then the motors start and the metals pour into the mold.

   If only one or two of the metals is at the appropriate temperature, Steps 5 and 6 are repeated until the metals can be successfully mixed.

8. The application unconfigures all of the transducers so that they no longer participate in parallel polls.

9. As a cleanup step before exiting, the application returns the interface board to its original state by taking it offline.

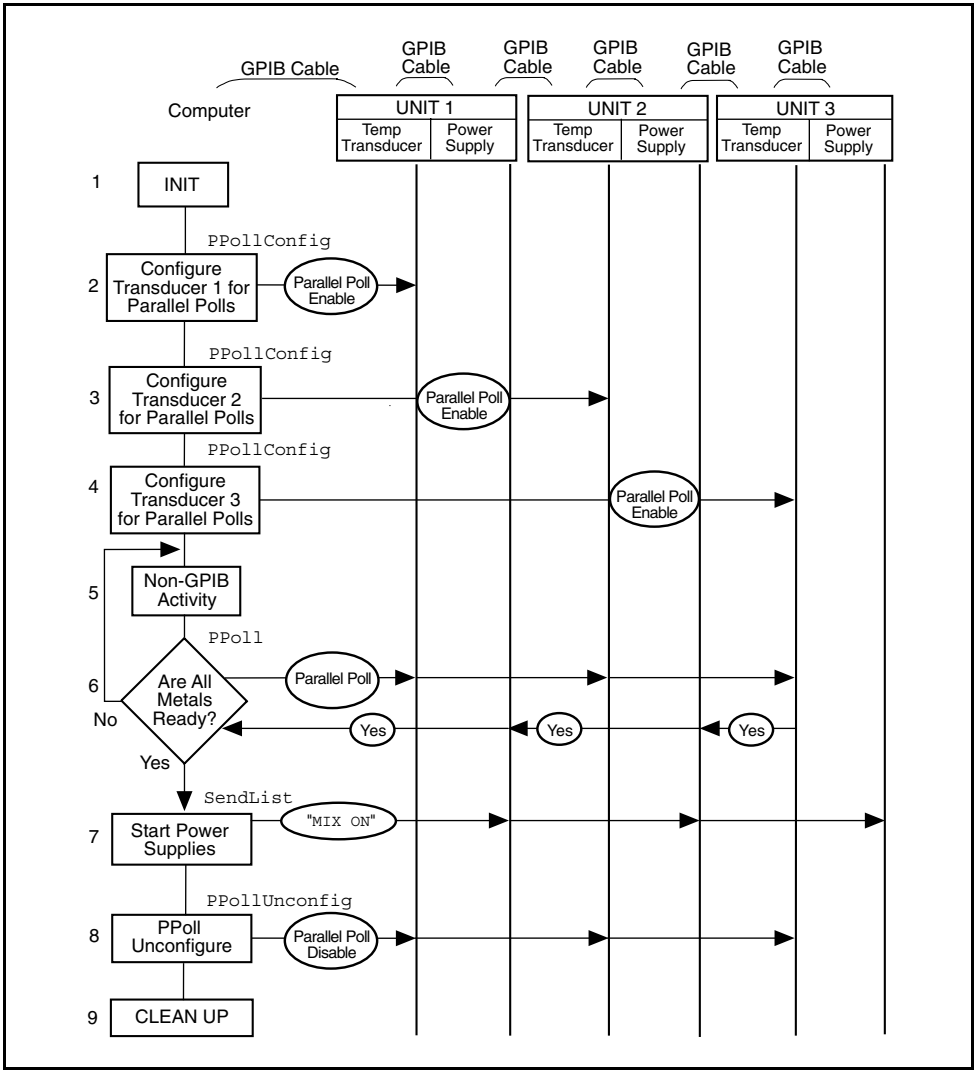



Figure 2-8. Program Flowchart for Example 8

## Example 9: Non-Controller Example

This example illustrates how you can use the NI-488.2 software to emulate a GPIB device that is not the GPIB Controller.

A software engineer has written firmware to emulate a GPIB device for a research project and is testing it using an application that makes simple GPIB calls. The following steps correspond to the program flowchart in Figure 2-9.

1. The application brings the device online.

2. The application waits for any of three events to occur: the device to become listen-addressed, become talk-addressed, or receive a GPIB clear message.

3. As soon as one of the events occurs, the application takes an action based upon the event that occurred. If the device was cleared, the application resets the internal state of the device to default values. If the device was talk-addressed, it writes data back to the Controller. If the device was listen-addressed, it reads in new data from the Controller.

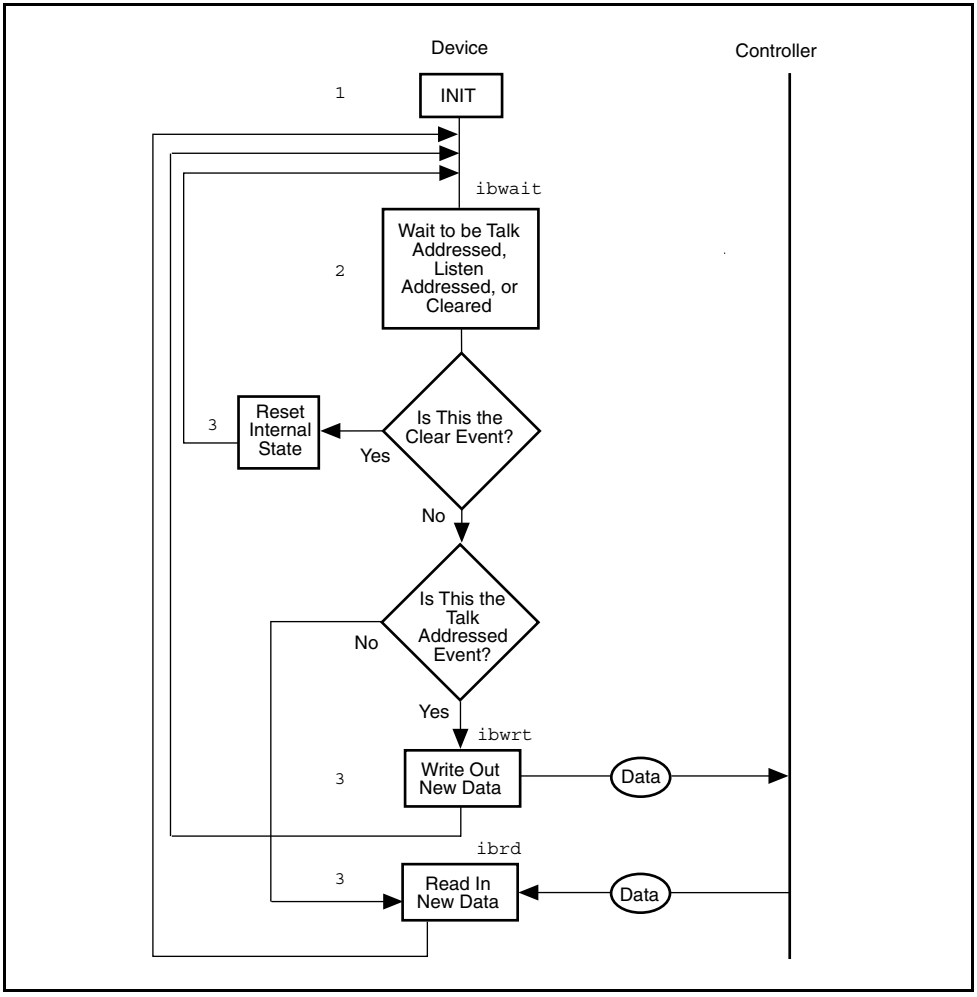



Figure 2-9. Program Flowchart for Example 9

# Chapter 3
# Developing Your Application

This chapter explains how to develop a GPIB application program using NI-488 functions and NI-488.2 routines.

## Choosing a Programming Method

Programs that need to communicate across the GPIB can access the NI-488.2 driver using either the NI-488.2 language interface or the universal language interface (ULI). If you are writing a new GPIB application program, use the NI-488.2 language interface.

### Using the NI-488.2 Language Interface

Your NI-488.2 software includes two distinct sets of subroutines to meet your application needs. For most application programs, the NI-488 functions are sufficient. You should use the NI-488.2 routines if you have a complex configuration with one or more interface boards and multiple devices.

The following sections discuss some differences between NI-488 functions and NI-488.2 routines.

#### Using NI-488 Functions: One Device for Each Board

If your system has only one device attached to each board, the NI-488 functions are probably sufficient for your programming needs. Some other factors that make the NI-488 functions more convenient include the following:

- With NI-488 asynchronous I/O functions (`ibcmda`, `ibrda`, and `ibwrta`), you can initiate an I/O sequence while maintaining control over the CPU for non-GPIB tasks.
- NI-488 functions include built-in file transfer functions (`ibrdf` and `ibwrtf`).
- With NI-488 functions, you can control the bus in non-typical ways or communicate with non-compliant devices.

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 more control over the GPIB than NI-488.2 routines. The following sections describe these different function types.

NI-488 Device Functions

Device functions are high-level functions that automatically execute commands that handle bus management operations such as reading from and writing to devices or polling them for status. If you use device functions, you do not need to understand GPIB protocol or bus management. For information about device-level calls and how they manage the GPIB, refer to *Device-Level Calls and Bus Management*, in Chapter 7, *GPIB Programming Techniques.*

NI-488 Board Functions

Board functions are low-level functions that perform rudimentary GPIB operations. Board functions access the interface board directly and require you to handle the addressing and bus management protocol. In cases when the high-level device functions might not meet your needs, low-level board functions give you the flexibility and control to handle situations such as the following:

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

The NI-488 board functions are compatible with, and can be interspersed within, sequences of NI-488.2 routines. When you use board functions within a sequence of NI-488.2 routines, you do not need a prior call to `ibfind` to obtain a board descriptor. You simply substitute the board index as the first parameter of the board function call. With this flexibility, you can handle non-standard or unusual situations that you cannot resolve using NI-488.2 routines only.

**Using NI-488.2 Routines: Multiple Boards and/or Multiple Devices**

When your system includes a board that must access more than one device, use the NI-488.2 routines. NI-488.2 routines can perform the following tasks with a single call:

- Find all of the Listeners on the bus
- Find a device requesting service
- Determine the state of the SRQ line, or wait for SRQ to be asserted
- Address multiple devices to listen

### Using the Universal Language Interface (ULI)

If you are writing a new GPIB application, use the NI-488.2 language interface. If you already have an existing application that uses HP-style calls, you can use the universal language interface (ULI) to access the NI-488.2 software. For more information about using the ULI, refer to Appendix C, *Universal Language Interface*.

## Checking Status with Global Variables

Each NI-488 function and NI-488.2 routine updates the global variables to reflect the status of the device or board that you are using. The status word (`ibsta`), the error variable (`iberr`), and the count variables (`ibcnt` and `ibcntl`) contain useful information about the performance of your application program. Your program should check these variables frequently. The following sections describe each of these global variables and how you can use them in your application program. You can print out the values of the global variables at any time while the application is running.

### Status Word – ibsta

All functions update a global status word, `ibsta`, which contains information about the state of the GPIB and the GPIB hardware. Most of the NI-488 functions return the value stored in `ibsta`. You can test for conditions reported in `ibsta` to make decisions about continued processing, or you can debug your program by checking `ibsta` after each call.

`ibsta` is a 16-bit value. A bit value of one (1) indicates that a certain condition is in effect. A bit value of zero (0) indicates that the condition is not in effect. Each bit in `ibsta` can be set for NI-488 device calls (dev), NI-488 board calls and NI-488.2 calls (brd), or both (dev, brd).

Table 3-1 shows the condition that each bit position represents, the bit mnemonics, and the type of calls for which each bit can be set. For a detailed explanation of each of the status conditions, refer to Appendix A, *Status Word Conditions*.

Table 3-1. Status Word (ibsta) Layout

| Mnemonic | Bit Pos. | Hex Value | 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 |
| SPOLL | 10 | 400 | brd | Board has been serial polled by Controller |
| EVENT | 9 | 200 | brd | DCAS, DTAS, or IFC event has occurred |
| 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 |

The language header files included on your distribution disk contain the mnemonic constants for `ibsta`. You can check a bit position in `ibsta` by using its numeric value or its mnemonic constant. For example, bit position 15 (hex 8000) detects a GPIB error. The mnemonic for this bit is ERR. To check for a GPIB error, use either of the following statements after each NI-488 function and NI-488.2 routine as shown:

```
if (ibsta & ERR) gpiberr();
```

or

```
if (ibsta & 0x8000) gpiberr();
```

where `gpiberr()` is an error handling routine.

### Error Variable – iberr

If the ERR bit is set in the status word (`ibsta`), a GPIB error has occurred. When an error occurs, the error type is specified by the value in `iberr`.

**Note:** ***The value in*** `iberr` ***is meaningful as an error type only when the ERR bit is set, indicating that an error has occurred.***

For more information on error codes and solutions refer to Chapter 4, *Debugging Your Application*, or Appendix B, *Error Codes and Solutions.*

### Count Variables – ibcnt and ibcntl

The count variables are updated after each read, write, or command function. `ibcnt` is a 16-bit integer and `ibcntl` is a 32-bit integer. If you are reading data, the count variables indicate the number of bytes read. If you are sending data or commands, the count variables reflect the number of bytes sent.

In your application program, you can use the count variables to null-terminate an ASCII string of data received from an instrument. For example, if data is received in an array of characters, you can use `ibcntl` to null-terminate the array and print the measurement on the screen as follows:

```
char rdbuf[512];
ibrd (ud, rdbuf, 20L);
if (!(ibsta & ERR)){
    rdbuf[ibcntl] = '\0';
    printf ("Read:  %s\n", rdbuf);
}
else {
    error();
}
```

`ibcntl` is the number of bytes received. Data begins in the array at index zero (0); therefore, `ibcntl` is the position for the null character that marks the end of the string.

## Using ibic to Communicate with Devices

Before you begin writing your application program, you might want to use the `ibic` utility. With `ibic` (Interface Bus Interactive Control), you communicate with your instruments from the keyboard rather than from an application program. Before you develop your GPIB application, you can use `ibic` to learn how to communicate with your instruments and to determine your programming needs. For specific device communication instructions, refer to the user manual that came with your instrument. For information about using `ibic` and for detailed examples, refer to Chapter 5, *ibic– Interface Bus Interactive Control Utility.*

After you have learned how to communicate with your devices in `ibic`, you are ready to begin writing your application program.

# Writing Your NI-488 Application

This section discusses items you should include in your application program, general program steps, and an NI-488 example. In this manual, the example code is presented in C using the standard C language interface. The NI-488.2 software includes the source code for this example written in C (`devsamp.c`) and the source code for this example written in BASIC (`devsamp.bas`).

The NI-488.2 software also includes the source code for nine application examples, which are described in Chapter 2, *Application Examples*.

## Items to Include

- Include the GPIB header file. This file contains prototypes for the NI-488 functions and constants that you can use in your application program.
- Check for errors after each NI-488 function call.
- Declare and define a function to handle GPIB errors. This function takes the device offline and closes the application. If the function is declared as follows:

  ```
  void gpiberr (char *msg);    /* function prototype */
  ```

  then your application invokes the function as follows:

  ```
  if (ibsta & ERR) {
     gpiberr("GPIB error");
  }
  ```

## NI-488 Program Shell

Figure 3-1 is a flowchart of the steps to create your application program using device-level NI-488 functions.

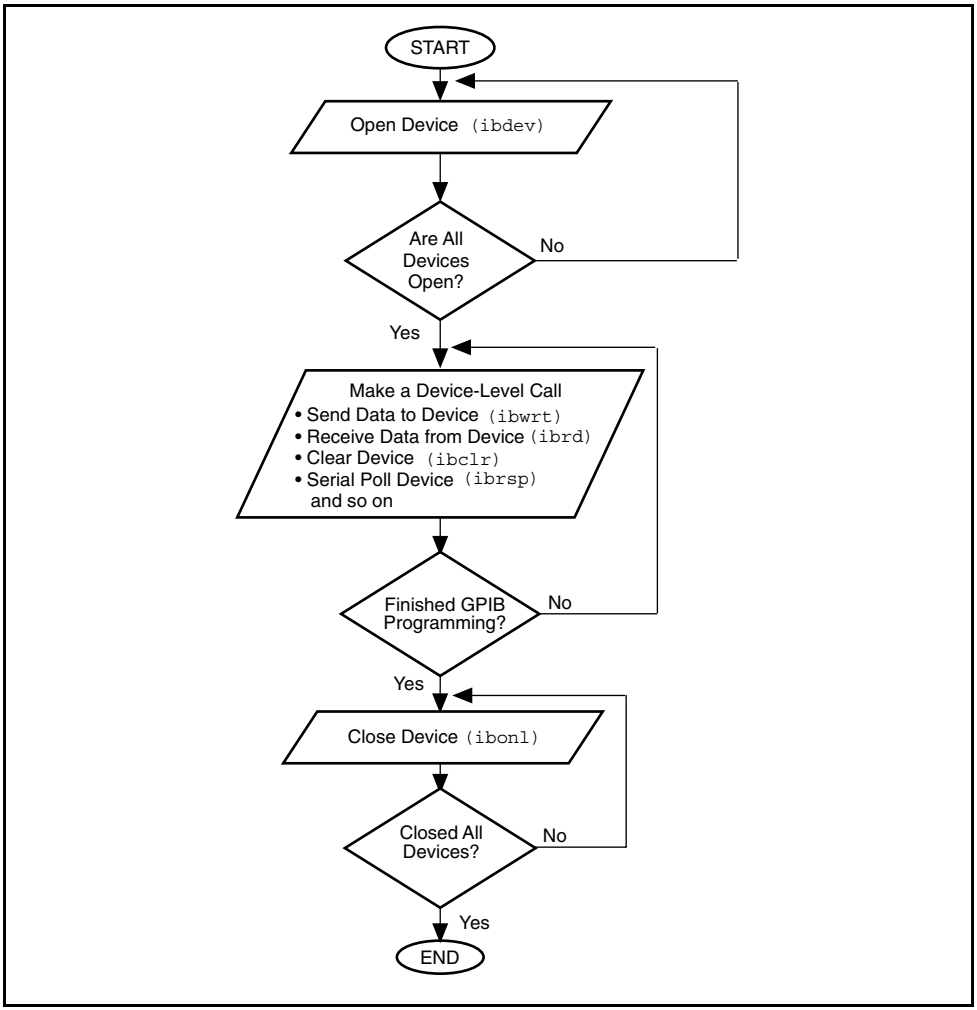



Figure 3-1. General Program Shell Using NI-488 Device Functions

## General Program Steps and Examples

The following steps demonstrate how to use the NI-488 device functions in your program. This example configures a digital multimeter, reads 10 voltage measurements, and computes the average of these measurements.

### Step 1. Open a Device

Your first NI-488 function call should be to `ibdev` to open a device.

```
ud = ibdev(0, 1, 0 , T10s, 1, 0);

if (ibsta & ERR) {
   gpiberr("ibdev error");
}
```

The input arguments of the `ibdev` function are as follows:

- 0 - board index for GPIB0
- 1 - primary GPIB address of the device
- 0 - no secondary GPIB address for the device
- T10s - I/O timeout value (10 s)
- 1 - send END message with the last byte when writing to device
- 0 - disable EOS detection mode

When you call `ibdev`, the driver automatically initializes the GPIB by sending an Interface Clear (IFC) message and placing the device in remote programming state.

### Step 2. Clear the Device

Clear the device before you configure the device for your application. Clearing the device resets its internal functions to a default state.

```
ibclr(ud);
if (ibsta & ERR) {
   gpiberr("ibclr error");
}
```

### Step 3. Configure the Device

After you open and clear the device, it is ready to receive commands. To configure the instrument, you send device-specific commands using the `ibwrt` function. Refer to the instrument user manual for the command bytes that work with your instrument.

```
ibwrt(ud, "*RST; VAC; AUTO; TRIGGER 2; *SRE 16", 35L);
if (ibsta & ERR) {
   gpiberr("ibwrt error");
}
```

The programming instruction in this example resets the multimeter (`*RST`). The meter is instructed to measure the volts alternating current (`VAC`) using auto-ranging (`AUTO`), to wait for a trigger from the GPIB interface board before starting a measurement (`TRIGGER 2`), and to assert the SRQ line when the measurement completes and the multimeter is ready to send the result (`*SRE 16`).

### Step 4. Trigger the Device

If you configure the device to wait for a trigger, you must send a trigger command to the device before reading the measurement value. Then instruct the device to send the next triggered reading to its GPIB output buffer.

```
ibtrg(ud);
if (ibsta & ERR) {
   gpiberr("ibtrg error");
}

ibwrt(ud,"VAL1?", 5L);
if (ibsta & ERR) {
   gpiberr("ibwrt error");
}
```

### Step 5. Wait for the Measurement

After you trigger the device, the RQS bit is set when the device is ready to send the measurement. You can detect RQS by using the `ibwait` function. The second parameter indicates what you are waiting for. Notice that the `ibwait` function also returns when the I/O timeout value is exceeded.

```
printf("Waiting for RQS...\n");
ibwait (ud, TIMO | RQS);
if (ibsta & (ERR | TIMO)) {
   gpiberr("ibwait error");
}
```

When SRQ has been detected, serial poll the instrument to determine if the measured data is valid or if a fault condition exists. For IEEE 488.2 instruments, you can find out by checking the message available (MAV) bit, bit 4 in the status byte that you receive from the instrument.

```
ibrsp (ud, &StatusByte);
if (ibsta & ERR) {
   gpiberr("ibrsp error");
}

if ( !(StatusByte & MAVbit)) {
   gpiberr("Improper Status Byte");
   printf("  Status Byte = 0x%x\n", StatusByte);
}
```

**Step 6. Read the Measurement**

If the data is valid, read the measurement from the instrument. (`AsciiToFloat` is a function that takes a null-terminated string as input and outputs the floating point number it represents.)

```
ibrd (ud, rdbuf, 10L);
if (ibsta & ERR) {
   gpiberr("ibrd error");
}

rdbuf[ibcntl] = '\0';
printf("Read: %s\n", rdbuf);
      /*  Output ==> Read: +10.98E-3  */

sum += AsciiToFloat(rdbuf);
```

**Step 7. Process the Data**

Repeat steps 4 through 6 in a loop until 10 measurements have been read. Then print the average of the readings as shown:

```
printf("The average of the 10 readings is %f\n", sum/10.0);
```

**Step 8. Place the Device Offline**

As a final step, take the device offline using the `ibonl` function.

```
ibonl (ud, 0);
```

# Writing Your NI-488.2 Application

This section discusses items you should include in an application program that uses NI-488.2 routines, general program steps, and an NI-488.2 example. In this manual the example code is presented in C using the standard C language interface. The NI-488.2 software includes the source code for this example written in C (`samp4882.c`) and the source code for this example written in BASIC (`samp488.2.bas`).

The NI-488.2 software also includes the source code for nine application examples, which are described in Chapter 2, *Application Examples*.

## Items to Include

- Include the appropriate GPIB header file. This file contains prototypes for the NI-488.2 routines and constants that you can use in your application program.
- Check for errors after each NI-488.2 routine.
- Declare and define a function to handle GPIB errors. This function takes the device offline and closes the application. If the function is declared as follows:

  ```
  void gpiberr (char *msg);    /* function prototype */
  ```

  then your application invokes the function as follows:

  ```
  if (ibsta & ERR) {
     gpiberr("GPIB error");
  }
  ```

## NI-488.2 Program Shell

Figure 3-2 is a flowchart of the steps to create your application program using NI-488.2 routines.

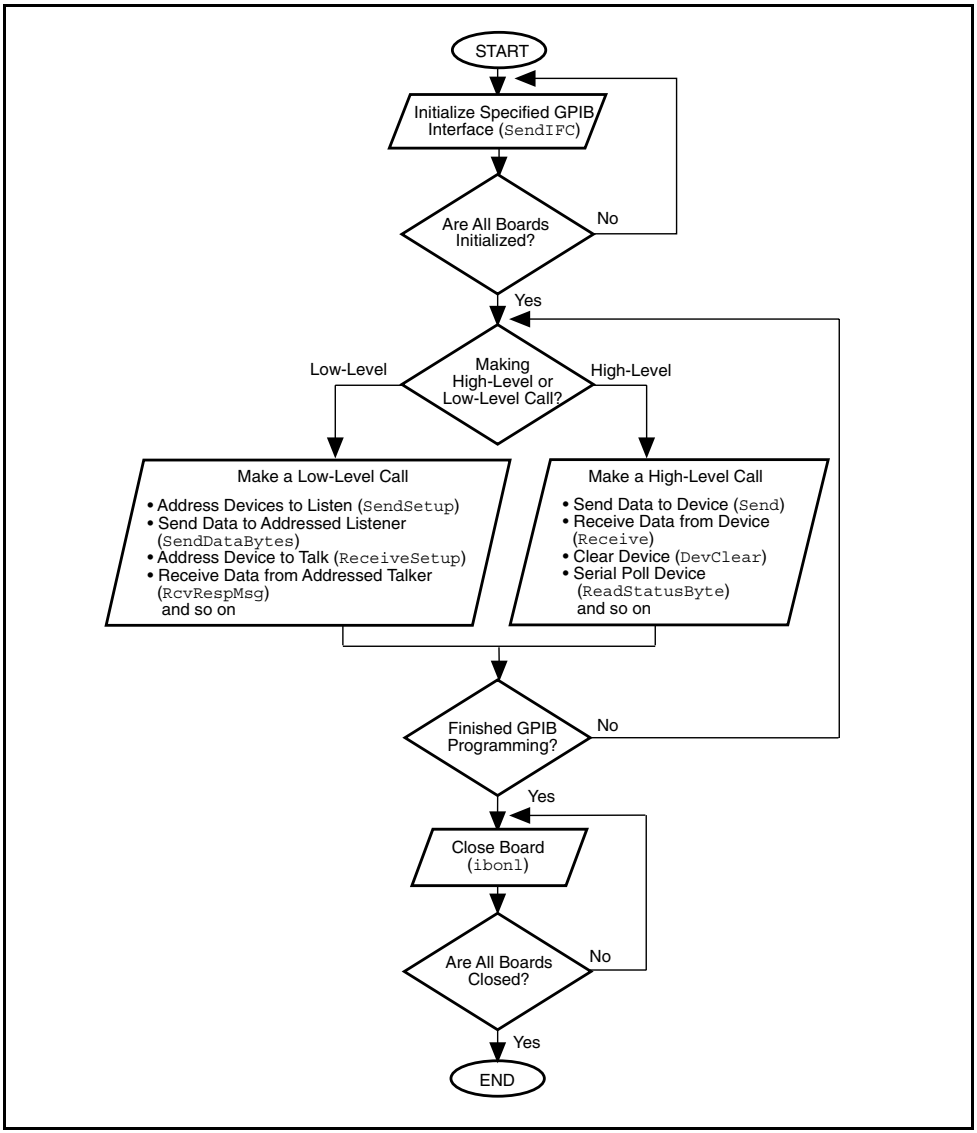



Figure 3-2. General Program Shell Using NI-488.2 Routines

## General Program Steps and Examples

The following steps demonstrate how to use the NI-488.2 routines in your program. This example configures a digital multimeter, reads 10 voltage measurements, and computes the average of these measurements.

### Step 1. Initialization

Use the `SendIFC` routine to initialize the bus and the GPIB interface board so that the GPIB board is Controller-In-Charge (CIC). The only argument of `SendIFC` is the GPIB interface board number.

```
SendIFC(0);
if (ibsta & ERR) {
   gpiberr("SendIFC error");
}
```

### Step 2. Find All Listeners

Use the `FindLstn` routine to create an array of all of the instruments attached to the GPIB. The first argument is the interface board number, the second argument is the list of instruments that was created, the third argument is a list of instrument addresses that the procedure actually found, and the last argument is the maximum number of devices that the procedure can find (that is, it must stop if it reaches the limit). The end of the list of addresses must be marked with the `NOADDR` constant, which is defined in the header file that you included at the beginning of the program.

```
for (loop = 0; loop <=30; loop++){
   instruments[loop] = loop;
}
instruments[31] = NOADDR;

printf("Finding all Listeners on the bus...\n");

Findlstn(0, instruments, result, 30);
if (ibsta & ERR) {
   gpiberr("FindLstn error");
}
```

### Step 3. Identify the Instrument

Send an identification query to each device for identification. For this example, assume that all of the instruments are IEEE 488.2-compatible and can accept the identification query, `*IDN?`. In addition, assume that `FindLstn` found the GPIB interface board at primary address 0 (default) and, therefore, you can skip the first entry in the `result` array.

```
for (loop = 1; loop <= num_Listeners; loop++) {
    Send(0, result[loop], "*IDN?", 5L, NLend);
    if  (ibsta & ERR) {
        gpiberr("Send error");
    }

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

    buffer[ibcntl] = '\0';
    printf("The instrument at address %d is a %s\n",
            result[loop], buffer);
    if (strncmp(buffer, "Fluke, 45", 9) == 0) {
        fluke = result[loop];
        printf("****  Found the Fluke ****\n");
        break;
    }
}

if (loop > num_Listeners) {
    printf("Did not find the Fluke!\n");
    ibonl(0,0);
    exit(1);
}
```

The constant `NLend` signals that the new line character with EOI is automatically appended to the data to be sent.

The constant `STOPend` indicates that the read is stopped when EOI is detected.

### Step 4. Initialize the Instrument

After you find the multimeter, use the `DevClear` routine to clear it. The first argument is the GPIB board number. The second argument is the GPIB address of the multimeter. Then send the IEEE 488.2 reset command to the meter.

```
DevClear(0, fluke);
if (ibsta & ERR) {
    gpiberr("DevClear error")
}

Send(0, fluke, "*RST", 4L, NLend);
if  (ibsta & ERR) {
    gpiberr("Send *RST error");
}
sum = 0.0;
for(m =0; m<10; m++){
/* start of loop for Steps 5 through 8 */
```

**Step 5. Configure the Instrument**

After initialization, the instrument is ready to receive instructions. To configure the multimeter, use the `Send` routine to send device-specific commands. The first argument is the number of the access board. The second argument is the GPIB address of the multimeter. The third argument is a string of bytes to send to the multimeter.

The bytes in this example 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 SRQ when the measurement has been completed and the meter is ready to send the result (`*SRE 16`). The fourth argument represents the number of bytes to be sent. The last argument, `NLend`, is a constant defined in the header file which tells `Send` to append a linefeed character, with EOI asserted, to the end of the message sent to the multimeter.

```
Send (0, fluke, "VAC; AUTO; TRIGGER 2; *SRE 16", 29L, NLend);
if (ibsta & ERR) {
   gpiberr("Send setup error");
}
```

**Step 6. Trigger the Instrument**

In the previous step, the multimeter was instructed to wait for a trigger before conducting a measurement. Now send a trigger command to the multimeter. You could use the `Trigger` routine to accomplish this, but because the Fluke 45 is IEEE 488.2-compatible, you can just send it the trigger command, `*TRG`. The `VAL1?` command instructs the meter to send the next triggered reading to its output buffer.

```
Send(0, fluke, "*TRG; VAL1?", 11L, NLend);
if (ibsta & ERR) {
   gpiberr("Send trigger error");
}
```

**Step 7. Wait for the Measurement**

After the meter is triggered, it takes a measurement and displays it on its front panel and then asserts SRQ. You can detect the assertion of SRQ using either the `TestSRQ` or `WaitSRQ` routine. If you have a process that you want to execute while you are waiting for the measurement, use `TestSRQ`. For this example, you can use the `WaitSRQ` routine. The first argument in `WaitSRQ` is the GPIB board number. The second argument is a flag returned by `WaitSRQ` that indicates whether or not SRQ is asserted.

```
WaitSRQ(0, &SRQasserted);
if (!SRQasserted) {
   gpiberr("WaitSRQ error");
}
```

After you have detected SRQ, use the `ReadStatusByte` routine to poll the meter and determine its status. The first argument is the GPIB board number, the second argument is the GPIB address of the instrument, and the last argument is a variable that `ReadStatusByte` uses to store the status byte of the instrument.

```
ReadStatusByte(0, fluke, &statusByte);
if (ibsta & ERR) {
   gpiberr("ReadStatusByte error");
}
```

After you have obtained the status byte, you must check to see if the meter has a message to send. You can do this by checking the message available (MAV) bit, bit 4 in the status byte.

```
if (!(statusByte & MAVbit) {
   gpiberr("Improper Status Byte");
   printf("Status Byte = 0x%x\n", statusByte);
}
```

**Step 8. Read the Measurement**

Use the `Receive` function to read the measurement over the GPIB. The first argument is the GPIB interface board number, and the second argument is the GPIB address of the multimeter. The third argument is a string into which the `Receive` function places the data bytes from the multimeter. The fourth argument represents the number of bytes to be received. The last argument indicates that the `Receive` message terminates upon receiving a byte accompanied with the END message.

```
Receive(0, fluke, buffer, 10L, STOPend);
if (ibsta & ERR) {
   gpiberr("Receive error");
}

buffer[ibcntl] = '\0';
printf (Reading : %s\n", buffer);
sum += AsciiToFloat(buffer);
}  /*  end of loop started in Step 5 */
```

**Step 9. Process the Data**

Repeat Steps 5 through 8 in a loop until 10 measurements have been read. Then print the average of the readings as shown:

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

**Step 10. Place the Board Offline**

Before ending your application program, take the board offline using the `ibonl` function.

```
ibonl(0,0);
```

## Compiling, Linking, and Running Your C Application

This section describes how to compile, link, and run your application program. To see examples that show how to include specific lines of code in your program, refer to the program examples included with your NI-488.2 software.

Before you compile your application program, make sure that the following line is included at the beginning of your program:

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

After you have written your Microsoft C application program, you need to compile your application program using the Microsoft C (version 5.1 or higher) compiler. Then link your application with the C language interface, `mcib.lib`. Use the following command to compile and link a C application named `cprog` using Microsoft C:

```
cl cprog.c mcib.lib
```

After you have written your Borland C++ application, you need to compile your application program using the Borland C++ (version 2.0 or higher) compiler. Then link your application with the C language interface, `mcib.lib`. Use the following command to compile and link an application named `cprog` using Borland C++:

```
bcc cprog.c mcib.lib
```

To run your Microsoft C or Borland C++ application, enter the following command:

```
cprog
```

If you discover errors when you execute the program, refer to Chapter 4, *Debugging Your Application*. If you want to verify the NI-488 and NI-488.2 calls made by your application, you can use `appmon`, the GPIB Applications Monitor described in Chapter 6, *appmon–GPIB Applications Monitor.*

# Compiling, Linking, and Running Your BASIC Application

The NI-488.2 software includes language interfaces for Microsoft Professional BASIC (version 7.0 or higher), Microsoft Visual Basic (version 1.0), QuickBASIC (version 4.0 or higher), BASICA, and GWBASIC.

The following sections describe how to compile, link, and run your application program. To see examples that show how to include specific lines of code in your program, refer to the program examples included with your NI-488.2 software.

With the BASIC languages, you access the NI-488 functions as subroutines, using the BASIC keyword CALL followed by the NI-488 function name (for example `CALL ibtrg (ud%)`). With QuickBASIC, BASIC, or Visual Basic for DOS, you can access the NI-488 functions using another set of functions, the `il` functions (for example `result% = iltrg (ud%)`). If you prefer calling functions instead of subroutines, then you can use the `il` functions. For the BASIC languages, with some of the `ib` commands (for example, `ibrd`) the length of the string buffer is automatically calculated within the actual subroutine, which eliminates the need to pass in the length as an extra parameter.

If you discover errors when you execute the program, refer to Chapter 4, *Debugging Your Application*. If you want to verify the NI-488 and NI-488.2 calls made by your application, you can use `appmon`, the GPIB Applications Monitor described in Chapter 6, *appmon–GPIB Applications Monitor*.

## Microsoft BASIC

Before you compile your application program, make sure that the following line is included at the beginning of your program:

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

After you have written your BASIC application program, you need to compile it using the Microsoft Professional BASIC (version 7.0 or higher) compiler. Then link your application with the BASIC language interface, `mbib.obj`.

### Using the QBX Environment

Use the following commands to compile, link, and run a BASIC program named `bprog` from within the QBX environment.

To create an object module library called `mbib.lib`, enter the following command on the DOS command line:

```
lib mbib.lib + mbib.obj;
```

To create a QuickLibrary called `mbib.qlb` that is linked with the `mbib.obj` language interface, enter the following command on the DOS command line:

```
link /q mbib.obj, mbib.qlb,, qbxqlb.lib;
```

To run the program from within the QBX environment, type the following on the DOS command line:

```
qbx bprog /l mbib.qlb
```

Then select **Start** from the **Run** menu.

#### Using the DOS Command Line

Use the following commands to compile, link, and run a BASIC program named `bprog` from the DOS command line.

To compile the BASIC program, enter the following command on the DOS command line:

```
bc bprog;
```

To link the BASIC program with the `mbib.obj` language interface, enter the following command on the DOS command line:

```
link bprog mbib;
```

To run the program from the DOS command line, type:

```
bprog
```

## Microsoft Visual Basic

Before you compile and link your program in Visual Basic, you must include the following line in the module-level code:

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

Do not list `mbdecl.bas` in a project `.mak` file.

Use the following command to prepare the BASIC language interface, `mbib.obj`, for the Visual Basic environment:

```
lib vbib.lib + mbib.obj + vbdos.lib;
link /q mbib.obj vbdos.lib,vbib.qlb,,vbdosqlb.lib;
```

Enter the following command to run a Visual Basic program named `bprog` in the Visual Basic environment:

```
vbdos bprog /l vbib.qlb
```

Then select the **Start** option from the **Run** menu.

## QuickBASIC

Before you compile your application program, make sure that the following line is included at the beginning of your program:

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

After you have written your QuickBASIC application program, you need to compile it using the Microsoft QuickBASIC (version 4.0 or higher) compiler. Then link your application with the QuickBASIC language interface, `qbib.obj`.

### Using the QuickBASIC Interactive Environment

Use the following commands to compile, link, and run a QuickBASIC program named `bprog` from the QuickBASIC interactive environment.

To create an object module library called `qbib.lib`, enter the following command on the DOS command line:

```
lib qbib.lib + qbib.obj;
```

To create a QuickLibrary called `qbib.qlb` that is linked with the `qbib.obj` language interface, enter one of the following commands on the DOS command line.

For QuickBASIC 4.0 compilers, type:

```
link /q qbib.obj, qbib.qlb,, bqlb40.lib;
```

or

For QuickBASIC 4.5 compilers, type:

```
link /q qbib.obj, qbib.qlb,, bqlb45.lib;
```

To run the program from within the QuickBASIC interactive environment, type the following on the DOS command line:

```
qb bprog /l qbib.qlb
```

Then select **Start** from the **Run** menu.

**Using the DOS Command Line**

Use the following commands to compile, link, and run a QuickBASIC program named `bprog` from the DOS command line.

To compile the BASIC program, enter the following command on the DOS command line:

```
bc bprog;
```

To link the BASIC application program with the `qbib.obj` language interface, enter the following command on the DOS command line:

```
link bprog qbib;
```

To run the program from the DOS command line, type:

```
bprog
```

## BASICA/GWBASIC

The BASICA language interface, `bib.m`, must be in the directory currently in use.

Use the following code to run a BASICA program named `bprog`:

```
load "bprog.bas
merge "decl.bas
run
```

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 most users, this constant is correct. But, if you invoke BASICA and it reports that there are fewer than 59 KB free, you need to adjust the constant to a smaller value.

# Chapter 4
# Debugging Your Application

This chapter describes several ways to debug your application program.

## Running ibtest

Before you run your application program, you should run the software diagnostic test, `ibtest`, that came with your NI-488.2 software. The `ibtest` program is an NI-488.2 application that makes calls to the driver. If `ibtest` passes, your GPIB hardware and NI-488.2 software are interacting correctly. The following paragraphs describe the messages you might receive while running `ibtest`, and how to resolve each problem. The term `gpibx` refers to one of the boards `gpib0`, `gpib1`, `gpib2`, and `gpib3`.

### Presence Test of Driver

The `ibtest` program tests for the presence of the NI-488.2 driver and displays the following message if it detects a problem:

**<<< No driver present for GPIB*x*. >>>**

If this message appears, make sure that the GPIB driver is installed. Check that the following line is in your `config.sys` file:

`device = `*`drive`*`:\`*`path`*`\gpib.com`

where *`drive`* is the driver where the NI-488.2 software is installed (usually `c`) and `path` is the path on the drive to the NI-488.2 software (for example, `at-gpib`).

### Presence Test of Board

The following error message appears if `gpib`*`x`* is not installed or if the software is not configured properly:

**<<< No board present for GPIB*x*. >>>**

If this message appears, you could have one of the following situations:

- The `Use this GPIB Interface` field in `ibconf` might be set to `no` for board `GPIB`*`x`*. If you want to use the board, you must set this field to `yes`.
- The board might not be properly installed and configured. Refer to the getting started manual for detailed instructions.

- The software and hardware settings do not match. You can run `ibconf` to check the current configuration of the software.

### GPIB Cables Connected

The following error message appears if a GPIB cable is connected to the board when you run `ibtest`:

```
Call(25) 'ibcmd " "' failed, ibsta (0x134) not what was expected (0x8130)

Call(25) 'ibcmd " "' failed, expected ibsta (0x100) to have the ERR bit
set.
```

Disconnect all GPIB cables before trying the test again.

### ULI Driver Loaded

If you try to use NI-488 functions or NI-488.2 routines or run `ibtest` with the ULI driver `uli.com` loaded, the following error message appears and your computer might lock up.

```
Syntax Error
```

While the ULI driver is loaded, you cannot use the standard NI-488 functions or NI-488.2 routines. Reboot your computer so that the ULI driver is not loaded. If your `autoexec.bat` file loads `uli.com`, change the line that loads `uli.com` to a comment before rebooting your computer.

## Running GPIBInfo

The `GPIBInfo` utility program is a simple diagnostic tool you can use to obtain information about the NI-488.2 software you are using and any GPIB interface boards in your system. This information helps you determine the capabilities of your NI-488.2 software and is also helpful if you need to call National Instruments for technical support.

If you run GPIBInfo with no specific parameters, the program displays software information such as the name and version of your GPIB software, the type of GPIB interface board and functions that you can use with the software, and whether or not you can use the HS488 high-speed protocol. GPIBInfo also displays information about each GPIB interface board installed in your system, including the name of the board, its Controller chip, the hardware settings, the type of functions that you can use, and whether or not the board can use the HS488 high-speed communications protocol. The typical GPIBInfo output is as follows:

```
GPIBInfo (Sep 29 1993)
Copyright (c) 1993 National Instruments Corp.  All rights
reserved.

Software Information:
  The NI-488.2 Software for MS-DOS is loaded.
  You are running Version 2.5 for the AT-GPIB/TNT board.
  It supports both the NI-488 functions and the NI-488.2 routines.
  It supports the HS488 high-speed protocol.

Hardware Information:
  GPIB0: an AT-GPIB/TNT board using the TNT4882C chip.
        It supports both the NI-488 functions and NI-488.2
        routines.
        It supports the HS488 high-speed protocol.
        It uses base I/O address 0x2C0.
        It uses interrupt level 11.
        It uses DMA channel 5.
```

You can also run GPIBInfo with a single parameter. The parameter should be the base I/O address of a GPIB board in your system. If GPIBInfo finds a board at the given address, it displays information about that board. This feature of GPIBInfo is useful in determining whether you have a GPIB board installed at a particular address. If you have a board installed at base I/O address 2C0 (hex), entering `gpibinfo 0x2c0` produces the following output:

```
GPIBInfo (Sep 29 1993)
Copyright (c) 1993 National Instruments Corp.  All rights
reserved.

The board at base I/O address 0x2C0 appears to be an AT-GPIB/TNT.
  It uses the TNT4882C GPIB Controller chip.
  It supports both the NI-488 functions and the NI-488.2 routines.
  It supports the HS488 high-speed protocol.
  The NI-488.2 software is configured to access this board as
  GPIB0.
```

## Debugging with the Global Status Variables

After each function call to your NI-488.2 driver, `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are updated before the call returns to your application. You should check for an error after each GPIB call. Refer to Chapter 3, *Developing Your Application*, for more information about how to use these variables within your program to automatically check for errors.

After you determine which GPIB call is failing and note the corresponding values of the global variables, refer to Appendix A, *Status Word Conditions*, and Appendix B, *Error Codes and Solutions*. These appendixes will help you interpret the state of the driver.

## Debugging with ibic

If your application does not automatically check for and display errors, you can locate an error by using `ibic`. Simply issue the same functions or routines, one at a time as they appear in your application program. Because `ibic` returns the status values and error codes after each call, you should be able to determine which GPIB call is failing. For more information about `ibic`, refer to Chapter 5, *ibic–Interface Bus Interactive Control Utility*.

After you determine which GPIB call is failing and note the corresponding values of the global variables, refer to Appendix A, *Status Word Conditions*, and Appendix B, *Error Codes and Solutions*. These appendixes will help you interpret the state of the driver.

## Debugging with appmon

The NI-488.2 software includes `appmon`, an applications monitor utility that is useful as a debugging tool. You can use `appmon` to monitor the NI-488 and NI-488.2 calls made by your DOS application. You can configure `appmon` to suspend the execution of your application and display an information screen any time the error bit is set in `ibsta`. For more information about `appmon`, refer to Chapter 6, *appmon–GPIB Applications Monitor*.

## GPIB Error Codes

Table 4-1 lists the GPIB error codes. Remember that the error variable is meaningful only when the ERR bit in the status variable is set. For a detailed description of each error and possible solutions, refer to Appendix B, *Error Codes and Solutions*.

Table 4-1. GPIB Error Codes

| Error Mnemonic | iberr Value | Meaning |
|---|---|---|
| EDVR | 0 | DOS error |
| ECIC | 1 | Function requires GPIB board to be CIC |
| ENOL | 2 | No Listeners on the GPIB |
| 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 |
| 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 |

## Configuration Errors

If your hardware and software settings do not match, one of the following problems might occur:

- Application hangs on input or output functions
- Data is corrupted

If these problems occur, make sure that the GPIB hardware settings match the NI-488.2 software settings for the interrupt request level and the DMA channel. Refer to the getting started manual that came with your kit for information on hardware and software default settings. For instructions on how to view or modify the NI-488.2 software configuration, refer to Chapter 8, *ibconf–Interface Bus Configuration Utility*.

To test your hardware, use the `ibdiag` program. Refer to your getting started manual for more information on `ibdiag`.

Several applications require customized configuration of the GPIB driver. For example, you might want to terminate reads on a special end-of-string character, or you might require secondary addressing. In these cases, you can use either the `ibconf` utility to permanently reconfigure the driver or the NI-488 `ibconfig` function to programmatically modify the driver while your application is running.

**Note**: ***To change settings other than base I/O address, interrupt level, or DMA channel, National Instruments recommends using*** `ibconfig` ***instead of running the*** `ibconf` ***utility***.

If your application uses `ibconfig`, it will always work regardless of the previous configuration of the driver. Refer to the description of `ibconfig` in the *NI-488.2 Function Reference Manual for DOS/Windows* for more information.

# Timing Errors

If your application fails, but the same calls issued in `ibic` are successful, your program might be issuing the NI-488.2 calls too quickly for your device to process and respond to them. This problem can also result in corrupted or incomplete data.

A well-behaved IEEE 488 device should hold off handshaking and set the appropriate transfer rate. If your device is not well behaved, you can test for and resolve the timing error by single-stepping through your program and inserting finite delays between each GPIB call. One way to do this is to have your device communicate its status whenever possible. Although this method is not possible with many devices, it is usually the best option. Your delays will be controlled by the device and your application can adjust itself and work independently on any platform. Other delay mechanisms will probably cause varying delay times on different platforms.

# Communication Errors

## Repeat Addressing

Some devices require GPIB addressing before any GPIB activity. Devices adhering to the IEEE 488.2 standard should remain in their current state until specific commands are sent across the GPIB to change their state. You might need to configure your NI-488.2 driver to perform repeat addressing if your device does not remain in its currently addressed state. Refer to Chapter 8, *ibconf–Interface Bus Configuration Utility*, or to the description of `ibconfig` (option `IbcREADDR`) in the *NI-488.2 Function Reference Manual for DOS/Windows* for more information about reconfiguring your software.

### Termination Method

You should be aware of the data termination method that your device uses. By default, your NI-488.2 software is configured to send EOI on writes and terminate reads on EOI or a specific byte count. If you send a command string to your device and it does not respond, it might be because it does not recognize the end of the command. You might need to send a termination message such as <CR> <LF> after a write command as follows:

```
ibwrt(dev,"COMMAND\x0A\x0D",9);
```

## Common Questions

**Can I have the DOS and Windows drivers installed at the same time?**

Yes, there is nothing wrong with installing both. However, it is better not to access them at the same time.

**What do I do if `ibdiag` or `ibtest` fail with an error?**

Refer to the *Running ibtest* section of this chapter and to the section about `ibdiag` in the getting started manual for specific information about what might cause these tests to fail.

**How do I communicate with my instrument over the GPIB?**

Refer to the documentation that came from the instrument manufacturer. The command sequences you use are totally dependent on the specific instrument. The documentation for each instrument should include the GPIB commands you need to communicate with it. In most cases, NI-488 device-level calls are sufficient for communicating with instruments. Refer to Chapter 3, *Developing Your Application*, for more information.

**Can I use the NI-488 and NI-488.2 calls together in the same application?**

Yes, you can mix NI-488 functions and NI-488.2 routines.

**Can I use the same name for my application and a GPIB board or device listed in `ibconf`?**

No, DOS devices share the same name space that file and directory names use. DOS does not operate properly if you have a file or a directory name that conflicts with one of the GPIB board or device names. File extensions have no effect on this problem. For example, using both `gpib0` and `gpib0.txt` would cause a problem. By default, the names used by the DOS driver are `gpib0`, `gpib1`, `gpib2`, `gpib3`, and `dev1`, `dev2`, `dev3`, and so on through `dev32`.

**What can I do to check for errors in my GPIB application?**

Examine the value of `ibsta` after each NI-488 or NI-488.2 call. If a call fails, the ERR bit of `ibsta` is set and an error code is stored in `ibcnt`. For more information about global status variables, refer to Chapter 3, *Developing Your Application*.

**How do I use `ibic`?**

You can use `ibic` to practice communication with your instrument, troubleshoot problems, and develop your application program. For instructions, refer to Chapter 5, *ibic–Interface Bus Interactive Control Utility*.

**How can I determine which type of GPIB board I have installed?**

Run the `GPIBInfo` utility. If you run `GPIBInfo` without specifying any parameters, it returns information about the GPIB boards currently configured for use in your system. If you know the base I/O address of a GPIB interface board, you can enter it as a parameter for specific information. For example, `gpibinfo 2C0` returns information about the GPIB board at base I/O address 2C0.

**How can I determine which version of the NI-488.2 software I have installed?**

Run the `GPIBInfo` utility. If you run `GPIBInfo` without specifying any parameters, it provides information about the version of the NI-488.2 software currently installed.

**What information should I have before I call National Instruments?**

When you call National Instruments, you should have the results of the diagnostic tests `ibdiag` and `ibtest` and the output from the `GPIBInfo` utility. Also, make sure you have filled out the technical support form in Appendix D, *Customer Communication*.

# Chapter 5
# ibic–Interface Bus Interactive Control Utility

This chapter introduces you to ibic, the interactive control program that you can use to communicate with GPIB devices interactively.

## Overview

With the ibic program, you communicate with the GPIB devices through functions you enter at the keyboard. For specific information about how to communicate with your particular device, refer to the manual that came with the device. You can use ibic to practice communication with the instrument, troubleshoot problems, and develop your application program.

One way ibic helps you to learn about your instrument and to troubleshoot problems is by displaying the following information on your screen whenever you enter a command:

- The results of the status word (ibsta) in hexadecimal notation
- The mnemonic constant of each bit set in ibsta
- The mnemonic value of the error variable (iberr) if an error exists (the ERR bit is set in ibsta)
- The count value for each read, write, or command function
- The data received from your instrument

## Example Using NI-488 Functions

This section shows how you might use ibic to test a sequence of NI-488 device function calls. You do not need to remember the parameters that each function takes. If you enter the function name only, ibic prompts you for the necessary parameters.

1. To run ibic, change to the appropriate drive and directory (c:\at-gpib in this example). Then enter the command ibic. Your screen should appear as follows:

```
C:\AT-GPIB> ibic

National Instruments
IEEE-488 Interface Bus Interactive Control Program (IBIC)
Copyright (c) 1993 National Instruments Corp. Version 3.0 (DOS)
Version Date: May 28 1993 Version Time: 09:42:25
All rights reserved

Type 'help' for help or 'q' to quit

:
```

2. The following example shows how you could use `ibdev` to open a device, assign it to access board `gpib0`, choose a primary address of 6 with no secondary address, set a timeout of 10 s, enable the END message, and disable the EOS mode:

```
:ibdev
    enter board index: 0
    enter primary address: 6
    enter secondary address: 0
    enter timeout: 13
    enter 'EOI on last byte' flag: 1
    enter end-of-string mode/byte: 0
id = 32256

ud0:
```

   You could also input all the same information with the `ibdev` command as follows:

```
:ibdev 0 6 0 13 1 0
id = 32256

ud0:
```

3. Clear the device as follows:

```
ud0: ibclr
[0100] (cmpl)
```

4. Write the function, range, and trigger source instructions to your device. Refer to the instrument's user manual for the command bytes that work with your instrument.

```
ud0: ibwrt
    enter string: "*RST; VAC; AUTO; TRIGGER 2; *SRE 16"
[0100] (cmpl)
count: 35
```

   or

```
ud0: ibwrt "*RST; VAC; AUTO; TRIGGER 2; *SRE 16"
[0100] (cmpl)
count: 35
```

5. Trigger the device as follows:

```
ud0: ibtrg
[0100] (cmpl)
```

6. Wait for a timeout or for your device to request service. If the current timeout limit is too short, use `ibtmo` to change it. Use the `ibwait` command as follows:

```
ud0: ibwait
   enter wait mask: TIMO RQS
[0900] (rqs cmpl)
```

or

```
ud0: ibwait TIMO RQS
[0900] (rqs cmpl)
```

7. Read the serial poll status byte. This serial poll status byte varies depending on the device used.

```
ud0: ibrsp
[0100] (cmpl)
Poll: 0x40 (decimal : 64)
```

8. Use the read command to display the data on the screen both in hex values and their ASCII equivalents.

```
ud0: ibrd
   enter byte count: 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 2b 30     . 0 0 4 7 E + 0
0a 0a                       . .
```

or

```
ud0: 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 2b 30     . 0 0 4 7 E + 0
0a 0a                       . .
```

9. Place the device offline as follows:

```
ud0: ibonl
   enter value: 0
[0100] (cmpl)
```

or

```
ud0: ibonl 0
[0100] (cmpl)
```

10. Terminate the `ibic` program by entering `q` at the prompt.

# ibic Syntax

When you enter commands in `ibic`, you can either include the parameters, or the program prompts you for values. Some commands require numbers as input values. Others might require you to input a string.

## Number Syntax

You can enter numbers as hexadecimal, octal, or decimal integer.

*Hexadecimal numbers*–You must precede hex numbers by zero and x (for example, 0xD).
*Octal numbers*–You must precede octal numbers by zero only (for example, 015).
*Decimal numbers*–Enter the number only.

## String Syntax

You can enter strings in the following formats.

*ASCII character sequence*–You must enclose the entire sequence in quotation marks (for example, `"*tst"`). To include a quotation mark in a string, precede it with a backslash (for example, `"ab\"cd"`).

*Octal bytes*–You must use a backslash character followed by the octal value. For example, octal 40 is represented by `\40` and can be used in a string as `"ab\40cd"`.

*Hex bytes*–You must use a backslash character and an `x` followed by the hex value. For example, hex 40 is represented by `\x40` and can be used in a string as `"ab\x40cd"`.

*Special Symbols*–Some instruments require special termination or end-of-string (EOS) characters that indicate to the device that a transmission has ended. The two most common EOS characters are `\r` and `\n`. `\r` represents a carriage return character and `\n` represents a linefeed character. You can use these special characters to insert the carriage return and linefeed characters into a string, as in `"F3R5T1\r\n"`.

## Address Syntax

Many of the NI-488.2 routines have an address or address list parameter. An address is a 16-bit representation of the GPIB address of a device. The primary address is stored in the low byte and the secondary address, if any, is stored in the high byte. For example, a device at primary address 6 and secondary address 0x67 has an address of 0x6706. A `NULL` address is represented as 0xffff.

## ibic Syntax for NI-488 Functions

Table 5-1 and Table 5-2 summarize the syntax of NI-488 functions in `ibic`. `v` represents a number and `string` represents a string that you input. For more information about the function parameters, use the `ibic` help feature or refer to the *NI-488.2 Function Reference Manual for DOS/Windows*.

Table 5-1. Syntax for Device-Level NI-488 Functions in ibic

| Syntax | Description |
|---|---|
| `ibask mn` | Return configuration information where `mn` is a mnemonic for a configuration parameter or equivalent integer value |
| `ibbna brdname` | Change access board of device where `brdname` is symbolic name of new board |
| `ibclr` | Clear specified device |
| `ibconfig mn v` | Alter configurable parameters where `mn` is mnemonic for a configuration parameter or equivalent integer value |
| `ibdev v v v v v v` | Open an unused device. `ibdev` parameters are `board id`, `pad`, `sad`, `tmo`, `eos`, `eot` |
| `ibeos v` | Change/disable EOS message |
| `ibeot v` | Enable/disable END message |
| `ibln v v` | Check for presence of device on the GPIB at `pad`, `sad` |
| `ibloc` | Go to local |
| `ibonl v` | Place device online or offline |
| `ibpad v` | Change primary address |
| `ibpct` | Pass control |
| `ibppc v` | Parallel poll configure |
| `ibrd v` | Read data where `v` is the bytes to read |
| `ibrda v` | Read data asynchronously where `v` is the bytes to read |
| `ibrdf flname` | Read data to file where `flname` is pathname of file to read |
| `ibrpp` | Conduct a parallel poll |
| `ibrsp` | Return serial poll byte |
| `ibsad v` | Change secondary address |
| `ibstop` | Abort asynchronous operation |
| `ibtmo v` | Change/disable time limit |
| `ibtrg` | Trigger selected device |
| `ibwait mask` | Wait for selected event where `mask` is a hex, octal, or decimal integer or a mask bit mnemonic |
| `ibwrt string` | Write data |
| `ibwrta string` | Write data asynchronously |
| `ibwrtf flname` | Write data from a file where `flname` is pathname of file to write |

Table 5-2. Syntax for Board-Level NI-488 Functions in ibic

| Syntax | Description |
|---|---|
| `ibask mn` | Return configuration information where `mn` is a mnemonic for a configuration parameter or equivalent integer value |
| `ibcac v` | Become Active Controller |
| `ibcmd string` | Send commands |
| `ibcmda string` | Send commands asynchronously |
| `ibconfig mn v` | Alter configurable parameters where `mn` is mnemonic for a configuration parameter or equivalent integer value |
| `ibdma v` | Enable/disable DMA |
| `ibeos v` | Change/disable EOS message |
| `ibeot v` | Enable/disable END message |
| `ibevent` | Return the oldest recorded event |
| `ibfind udname` | Return unit descriptor where `udname` is the symbolic name of a board (for example, `gpib0`) |
| `ibgts v` | Go from Active Controller to standby |
| `ibist v` | Set/clear `ist` |
| `iblines` | Read the state of all GPIB control lines |
| `ibln v v` | Check for presence of device on the GPIB at `pad`, `sad` |
| `ibloc` | Go to local |
| `ibonl v` | Place device online or offline |
| `ibpad v` | Change primary address |
| `ibppc v` | Parallel poll configure |
| `ibrd v` | Read data where `v` is the bytes to read |
| `ibrda v` | Read data asynchronously where `v` is the bytes to read |
| `ibrdf flname` | Read data to file where `flname` is pathname of file to read |
| `ibrpp` | Conduct a parallel poll |
| `ibrsc v` | Request/release system control |
| `ibrsv v` | Request service |
| `ibsad v` | Change secondary address |
| `ibsic` | Send interface clear |
| `ibsre v` | Set/clear remote enable line |
| `ibstop` | Abort asynchronous operation |
| `ibtmo v` | Change/disable time limit |

(continues)

Table 5-2. Syntax for Board-Level NI-488 Functions in ibic (Continued)

| Syntax | Description |
|---|---|
| `ibtrap v v` | Change the `appmon` configuration |
| `ibwait mask` | Wait for selected event where `mask` is a hex, octal, or decimal integer or a mask bit mnemonic |
| `ibwrt string` | Write data |
| `ibwrta string` | Write data asynchronously |
| `ibwrtf flname` | Write data from a file where `flname` is pathname of file to write |

## ibic Syntax for NI-488.2 Routines

Table 5-3 summarizes the syntax of NI-488.2 routines in `ibic. v` represents a number and `string` represents a string. `address` represents an address, and `addrlist` represents a list of addresses separated by commas. For more information about the routine parameters, use the `ibic` help feature or refer to the *NI-488.2 Function Reference Manual for DOS/Windows*.

Table 5-3. Syntax for NI-488.2 Routines in ibic

| Routine Syntax | Description |
|---|---|
| `AllSpoll addrlist` | Serial poll multiple devices |
| `DevClear address` | Clear a device |
| `DevClearList addrlist` | Clear multiple devices |
| `EnableLocal addrlist` | Enable local control |
| `EnableRemote addrlist` | Enable remote control |
| `FindLstn padlist v` | Find all Listeners |
| `FindRQS addrlist` | Find device asserting SRQ |
| `PassControl address` | Pass control to a device |
| `PPoll` | Parallel poll devices |
| `PPollConfig address v v` | Configure device for parallel poll |
| `PPollUnconfig addrlist` | Unconfigure device for parallel poll |
| `RcvRespMsg v v` | Receive response message |

(continues)

Table 5-3. Syntax for NI-488.2 Routines in ibic (Continued)

| Routine Syntax | Description |
|---|---|
| `ReadStatusByte address` | Serial poll a device |
| `Receive address v v` | Receive data from a device |
| `ReceiveSetup address` | Receive setup |
| `ResetSys addrlist` | Reset multiple devices |
| `Send address string v` | Send data to a device |
| `SendCmds string` | Send command bytes |
| `SendDataBytes buffer v` | Send data bytes |
| `SendIFC` | Send interface clear |
| `SendList addrlist string v` | Send data to multiple devices |
| `SendLLO` | Put devices in local lockout |
| `SendSetup addrlist` | Send setup |
| `Set 488.2 v` | Enter into 488.2 mode for board v |
| `SetRWLS addrlist` | Put devices in remote with lockout state |
| `TestSys addrlist` | Cause multiple devices to perform self tests |
| `TestSRQ` | Test for service request |
| `Trigger address` | Trigger a device |
| `TriggerList addrlist` | Trigger multiple devices |
| `WaitSRQ` | Wait for service request |

## Status Word

In `ibic`, all NI-488 functions (except `ibfind` and `ibdev`) and NI-488.2 routines return the status word `ibsta` in two forms: a hex value in square brackets and a list of mnemonics in parentheses. In the following example, the status word is on the second line. It shows that the device function write operation completed successfully:

```
ud0: ibwrt "f2t3x"
[0100] (cmpl)
count: 5

ud0:
```

For more information about the status word, refer to Chapter 3, *Developing Your Application*.

# Error Information

If an NI-488 function or NI-488.2 routine completes with an error, `ibic` displays the relevant error mnemonic. In the following example, an error condition EBUS has occurred during a data transfer.

```
ud0: ibwrt "f2t3x"
[8100] (err cmpl)
error: EBUS
count: 1

ud0:
```

In this example, the addressing command bytes could not be transmitted to the device. This indicates that either `dev1` is powered off, or the GPIB cable is disconnected.

For a detailed list of the error codes and their meanings, refer to Chapter 4, *Debugging Your Application*.

# 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.

If one of the addresses in an address list of an NI-488.2 routine is invalid, then the error is EARG and `ibic` displays the index of the invalid address as the count.

The count has a different meaning depending on which NI-488 function or NI-488.2 routine is called. Refer to the function descriptions in the *NI-488.2 Function Reference Manual for DOS/Windows* for the correct interpretation of the count return.

# Common NI-488 Functions

## ibfind

Use the `ibfind` function to open a board. The following example opens `gpib0`.

```
:ibfind gpib0
id = 32000

gpib0:
```

`id` is the unit descriptor of the board. The prompt `gpib0` indicates that the board is open.

Any name you use with the `ibfind` function must be a valid symbolic name in the driver. For more information about valid names, refer to Chapter 8, *ibconf–Interface Bus Configuration Utility*.

## ibdev

The `ibdev` command initializes a device descriptor with the input information.

With `ibdev`, you specify the following values:

- Access board for the device
- Primary address
- Secondary address
- Timeout setting
- EOT mode
- EOS mode

The following example 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 s (`tmo`=13), the END message enabled (`eot` =1), and the EOS mode disabled (`eos`= 0).

```
:ibdev 0 6 0x67 13 1 0
id = 32256

ud0:
```

If you use `ibdev` without specifying parameters, `ibic` prompts you for the input parameters as shown in the following example:

```
:ibdev
       enter board index: 0
       enter primary address: 6
       enter secondary address: 0x67
       enter timeout: 13
       enter 'EOI on last byte' flag: 1
       enter end-of-string mode/byte: 0
id = 32256

ud0:
```

Three distinct errors can occur with the `ibdev` call:

- EDVR–No device is available, the board index entered refers to a nonexistent board (that is, not 0, 1, 2, or 3), or no driver is installed. The following example illustrates an EDVR error.

```
:ibdev 4 6 0x67 7 1 0
id = -1
[8000] (err)
error: EDVR (2)

:
```

- ENEB–The board index entered refers to a known board (such as 0), but the driver cannot find the board. In this case, run `ibconf` to verify that the base address of the board is set correctly and that the `Use This GPIB Interface` field is set to `yes`.

- EARG–One of the last five parameters is an invalid value. The `ibdev` call returns with a new prompt and the EARG error (invalid function argument). If the `ibdev` call returns with an EARG error, you must identify which parameter is incorrect and use the appropriate command to correct it. In the following example, `pad` has an invalid value. You can correct it with an `ibpad` call as shown:

```
:ibdev 0 66 0x67 7 1 0
id = 32256
[8100] (err cmpl)
error: EARG

ud0: ibpad 6
previous value: 16
```

## ibwrt

The `ibwrt` command sends data from one GPIB device to another. For example, to send the six character data string `F3R5T1` from the computer to a device, you enter the following string at the prompt as shown in the following example:

```
ud0: ibwrt "F3R5T1"
[0100] (cmpl)
count: 6
```

The returned status word contains the `cmpl` bit, which indicates a successful I/O completion. The byte count `6` indicates that all six characters were sent from the computer and received by the device.

## ibrd

The `ibrd` command causes a GPIB device to receive data from another GPIB device. The following example acquires data from the device and displays it on the screen in hex format and in its ASCII equivalent, along with the status word and byte count.

```
ud0: ibrd 20
[2100] (end cmpl)
count: 18
4e 44 43 56 28 30 30 30    N D C V 9 0 0 0
2e 30 30 34 37 45 2b 30    . 0 0 4 7 E + 0
0d 0a                      . .
```

# Common NI-488.2 Routines in ibic

## Set

You must use the `set` command before you can use NI-488.2 routines in `ibic`. The syntax for this form of the `set` command is as follows:

```
set 488.2  n
```

where *n* represents a board number (for example, `n`=0 for `gpib0`).

The `488.2` prompt indicates that you are in NI-488.2 mode on board `n`. The following example shows how to enter into 488.2 mode on board `gpib0`.

```
set 488.2 0

488.2 (0):
```

## Send and SendList

The `Send` routine sends data to a single GPIB device. You can use the `SendList` command to send data to multiple GPIB devices. For example, suppose you want to send the five character string `*IDN?` followed by the new line character with EOI. You want to send the message from the computer to the devices at primary address 2 and 17. To do this, enter the `SendList` command at the `488.2 (0)` prompt as shown in the following example:

```
488.2 (0): SendList 2, 17 "*IDN?" NLend
[0128] (cmpl cic tacs)
count: 6
```

The returned status word contains the `cmpl` bit, which indicates a successful I/O completion. The byte count `6` indicates that six characters, including the added new line, were sent from the computer and received by both devices.

## Receive

The `Receive` routine causes the GPIB board to receive data from another GPIB device. The following example acquires 10 data bytes from the device at primary address 5. It stops receiving data when 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 `ibic` program also displays the status word and the count of transferred bytes.

```
488.2 (0): Receive 5 10 STOPend
[2124] (end cmpl cic lacs)
count: 5
48 65 6c 6c 6f      Hello
```

# Auxiliary Functions

Table 5-4 summarizes the auxiliary functions that you can use in ibic.

Table 5-4. Auxiliary Functions in ibic

| Function | Description |
|---|---|
| set udname | Select active device or board where udname is the symbolic name of the new device or board (for example, dev1 or gpib0). Call ibfind or ibdev initially to open each board or device. |
| help [option] | Display help information where option is any NI-488 or NI-488.2 call. If you do not enter an option, a menu of options appears. |
| ! | Repeat previous function. |
| - | Turn display off. |
| + | Turn display on. |
| n* function | Execute function n times where function represents the correct ibic function syntax. |
| n* ! | Execute previous function n times. |
| $ filename | Execute indirect file where filename is the pathname of a file that contains ibic functions to be executed. |
| print string | Display string on screen where string is an ASCII character sequence, octal bytes, hex bytes, or special symbols. |
| buffer option | Set the type of display used for buffers. Valid options are full, brief, ascii, and off. The default is full. |
| e | Exit |
| q | Quit. |

## Set (Select Device or Board)

You can use the set command to select 488.2 mode or to communicate with a different device. The following example switches communication from using NI-488.2 routines for gpib0 to using a unit descriptor (ud0) previously acquired by an ibdev call.

```
488.2 (0): set ud0

ud0:
```

## Help (Display Help Information)

The help feature displays a menu of topics to choose from. Each topic lists relevant functions and other information. You can access help for a specific NI-488 function or NI-488.2 routine by typing `help` followed by the call name (for example, `help ibwrt`). Help describes the function syntax for all NI-488 functions and NI-488.2 routines.

## ! (Repeat Previous Function)

The `!` function repeats the most recent function executed. The following example issues an `ibsic` command and then repeats that same command.

```
gpib0: ibsic
[0130] (cmpl cic atn)

gpib0: !
[0130] (cmpl cic atn)
```

## - (Turn Display Off) and + (Turn Display On)

The `-` function turns off all screen output except for the prompt. This function is useful when you want to repeat any I/O function quickly without waiting for screen output to be displayed.

The `+` function turns the screen output on.

In the following example 24 consecutive letters of the alphabet are read from a device using three `ibrd` calls.

```
ud0: ibrd 8
[2100] (end cmpl)
count: 8
61 62 63 64 65 66 67 68    a b c d e f g h

ud0: -

ud0: ibrd 8

ud0: +

ud0: ibrd 8
[2100] (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 five times to the device described by `ud0`.

```
ud0 : 5*ibwrt "Hello"
```

In the following example, the word `Hello` is sent five times, 20 times, and then 10 more times.

```
ud0: 5*ibwrt "Hello"
ud0: 20* !
ud0: 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)

The `$` function reads a specified file and executes the `ibic` functions listed in that file, in sequence, as if they were entered in that order from the keyboard. The following example executes the `ibic` functions listed in the file `userfile`.

```
gpib0: $ userfile
```

The following example repeats the operation three times.

```
gpib0: 3*$ userfile
```

The display mode that is in effect before this function was executed can be changed by functions in the indirect file.

## Print (Display the ASCII String)

You can use the `print` function to echo a string to the screen. The following example shows how you can use ASCII or hex with the print command.

```
dev1: print "hello"
hello

dev1: print "and\r\n\x67\x6f\x6f\x64\x62\x79\x65"
and
goodbye
```

You can also use `print` to display comments from indirect files. The print string appears even if the display is suppressed with the `-` function.

## Buffer (Set Buffer Display Mode)

You can set the type of display used for buffers to control how much of the buffer is displayed. `full` displays the entire buffer, `brief` displays the first and last eight bytes of the buffer, `ascii` displays the entire buffer in ASCII mode, and `off` displays none of the buffer.

# Chapter 6
# appmon–GPIB Applications Monitor

This chapter explains how to install, configure, and use `appmon`, the GPIB applications monitor, which is a useful debugging tool.

## Overview

`appmon` is useful for debugging sequences of GPIB calls made by your application program. You can configure `appmon` to suspend program execution whenever a specific event occurs. When `appmon` suspends the execution, you see a pop-up screen containing status information such as return values, global variables, and error codes.

## Installing appmon

`appmon` is included with the NI-488.2 software as the file `appmon.exe`. To install it, enter the following command at the DOS prompt:

```
appmon
```

If the GPIB driver is not present or `appmon` has already been installed, you receive an error message.

`appmon` remains installed until you restart the system.

## Configuring appmon

`appmon` can *trap*, suspend program execution, when certain conditions occur. You can select the following conditions to trap the program:

- Suspend on return from every NI-488.2 routine and NI-488 function
- Suspend on functions that return an error indication
- Suspend on calls that return with selected bit patterns in the GPIB status word `ibsta`

To select the options, you specify one or more *mask* flags, which determine the events that are trapped, and a *monitor mode*, which determines the display when a trap occurs.

To set the trap options, you can call `ibtrap` from your application program or you can run the special utility program called `ibtrap.exe`. When you call `ibtrap` from your application program, use the correct syntax as shown in the *NI-488.2 Function Reference Manual for DOS/Windows*.

To use the `ibtrap.exe` utility, enter `ibtrap` at the DOS prompt. Then press <F5> to select the trap mask or <F6> to select the monitor mode. When selecting a mask or monitor mode, use the arrow keys and spacebar to make changes. Press <Enter> to save the changes or <Esc> to cancel.

Table 6-1 lists the `ibtrap` mask options.

Table 6-1. ibtrap Mask Options

| Mask Flag | Mask Option |
|---|---|
| `-all` | All GPIB calls |
| `-err` | GPIB error |
| `-timo` | Timeout |
| `-end` | GPIB board detected END or EOS |
| `-srqi` | SRQ on |
| `-rqs` | Device requesting service |
| `-spoll` | GPIB board was serial polled |
| `-event` | DTAS, DCAS, or IFC event has occurred |
| `-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 |

Table 6-2 lists the three `ibtrap` monitor mode options.

Table 6-2. ibtrap Monitor Mode Options

| Monitor Mode Flag | Monitor Mode |
|---|---|
| `-off` | Monitor off–no recording or trapping occurs |
| `-rec` | Monitor records all GPIB driver calls, but no trapping occurs |
| `-dis` | Monitor records all GPIB driver calls and displays information whenever a trap condition exists |

Calling `ibtrap` without any flags displays the valid flags and their current state and has no effect on the `appmon` configuration.

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.

  If you select the `-dis` option and you run an application program that places the PC monitor into graphics mode, `appmon` records all GPIB driver calls, but is not displayed until your monitor is returned to character mode.

- `ibtrap -off` Disable the applications monitor. No recording or trapping is performed.

## Using appmon

When the applications monitor is displayed, you can view the parameters of the current GPIB call, change the display and trap modes, and scan information about previous GPIB calls. Figure 6-1 shows the information that `appmon` records for each GPIB call.

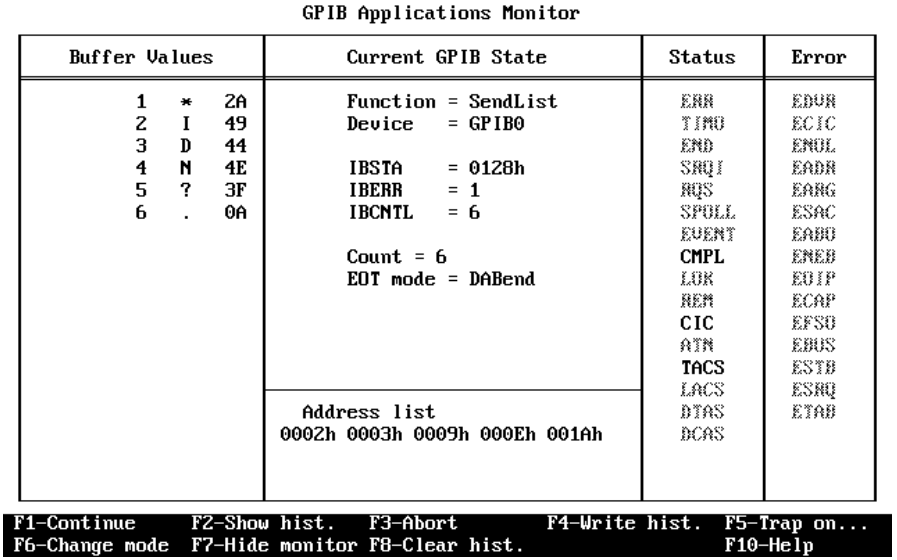



Figure 6-1. appmon Pop-up Screen

- `Function` NI-488.2 routine or NI-488 function description
- `Device` Symbolic device name
- `ibsta` GPIB status information
- `iberr` GPIB error information or the previous value of the `value` parameter if no error occurred
- `ibcntl` A 32-bit representation of the number of bytes transferred
- `Address List` Address list for functions that have an address list as a parameter
- `Buffer Value` Contents of the buffer for functions that have a buffer as a parameter–each byte of the buffer is shown with its index, character image, and ASCII value
- `Status` State of the individual bits of `ibsta`–all active bits are highlighted
- `Error` State of `iberr`–if an error occurred, the mnemonic for that error is highlighted

**Note:** `ibsta` ***values and ASCII codes for buffer values are always in hex. Other numbers are in decimal unless immediately followed by the letter h.***

## Using the Command Keys

Table 6-3 lists the function keys that you can use when the main `appmon` screen is displayed.

Table 6-3. Function Keys in appmon

| Key | Command |
|---|---|
| <F1> | Continue executing application program |
| <F2> | Display session summary |
| <F3> | Abort program execution and 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) |

## Viewing the GPIB History Screen

You can view information about previous GPIB calls by pressing <F2>. Once the information is displayed, use the cursor keys to manipulate the display. Press <Esc> or <F2> to exit the GPIB history display and return to the main `appmon` screen.

## Hiding and Showing the Applications Monitor

To hide the applications monitor and restore the contents of your screen, press <F7>. Press <F7> again to view the applications monitor pop-up screen.

# Chapter 7
# GPIB Programming Techniques

This chapter describes techniques for using some NI-488 functions and NI-488.2 routines in your application program.

For more detailed information about each function or routine, refer to the *NI-488.2 Function Reference Manual for DOS/Windows*.

## Termination of Data Transfers

GPIB data transfers are terminated either when the GPIB EOI line is asserted with the last byte of a transfer or when a preconfigured end-of-string (EOS) character is transmitted. By default, the NI-488.2 driver asserts EOI with the last byte of writes and the EOS modes are disabled.

You can use the `ibeot` function to enable or disable the end of transmission (EOT) mode. If EOT mode is enabled, the NI-488.2 driver asserts the GPIB EOI line when the last byte of a write is sent out on the GPIB. If it is disabled, the EOI line is *not* asserted with the last byte of a write.

You can use the `ibeos` function to enable, disable, or configure the EOS modes. EOS mode configuration includes the following information:

- An EOS byte
- EOS comparison method–This indicates whether the EOS byte has seven or eight significant bits. For a 7-bit EOS byte, the eighth bit of the EOS byte is ignored.
- EOS write method–If this is enabled, the NI-488.2 driver automatically asserts the GPIB EOI line when the EOS byte is written to the GPIB. For example, if the buffer passed into an `ibwrt` call contains five occurrences of the EOS byte, the EOI line is asserted as each of the five EOS bytes are written to the GPIB. If the `ibwrt` buffer does not contain an occurrence of the EOS byte, the EOI line is not asserted (unless the EOT mode is enabled, in which case the EOI line is asserted with the last byte of the write).
- EOS read method–If this is enabled, the NI-488.2 driver terminates `ibrd`, `ibrda`, and `ibrdf` calls when the EOS byte is detected on the GPIB, when the GPIB EOI line is asserted, or when the specified count is reached. If the EOS read method is disabled, `ibrd`, `ibrda`, and `ibrdf` calls terminate only when the GPIB EOI line is asserted or the specified count has been read.

You can use the `ibconfig` function to configure the software to inform you whether or not the GPIB EOI line was asserted when the EOS byte was read. Use the `IbcEndBitIsNormal` option to configure the software to report only the END bit in `ibsta` when the GPIB EOI line is asserted. By default, the NI-488.2 driver reports END in `ibsta` when either the EOS byte is read in or the EOI line is asserted during a read.

# High-Speed Data Transfers (HS488)

National Instruments has designed a high-speed data transfer protocol for IEEE 488 called *HS488*. This protocol increases performance for GPIB reads and writes up to 8 Mbytes/s, depending on your system.

HS488 is a superset of the IEEE 488 standard; thus, you can mix IEEE 488.1, IEEE 488.2, and HS488 devices in the same system. If HS488 is enabled, the TNT4882C hardware implements high-speed transfers automatically when communicating with HS488 instruments. To determine whether your GPIB interface board has the TNT4882C hardware, use the `GPIBInfo` utility. If you attempt to enable HS488 on a GPIB board that does not have the TNT4882C chip, the error ECAP is returned.

## Enabling HS488

To enable HS488 for your GPIB board, use the `ibconfig` function (option `IbcHSCableLength`). The value passed to `ibconfig` should specify the number of meters of cable in your GPIB configuration. If you specify a cable length that is much smaller than what you actually use, the transferred data could become corrupted. If you specify a cable length longer than what you actually use, the data is transferred successfully, but more slowly than if you specified the correct cable length.

In addition to using `ibconfig` to configure your GPIB board for HS488, the Controller-In-Charge must send out GPIB command bytes (interface messages) to configure other devices for HS488 transfers.

If you are using device-level calls, the NI-488.2 software automatically sends the HS488 configuration message to devices. If you enabled the HS488 protocol in `ibconf`, the NI-488.2 software sends out the HS488 configuration message when you use `ibdev` to bring a device online. If you call `ibconfig` to change the GPIB cable length, the NI-488.2 software sends out the HS488 message again the next time you call a device-level function.

If you are using board-level functions or NI-488.2 routines and you want to configure devices for high-speed, you must send the HS488 configuration messages using `ibcmd` or `SendCmds`. The HS488 configuration message is made up of two GPIB command bytes. The first byte, the Configure Enable (CFE) message (hex 1F), places all HS488 devices into their configuration mode. Non-HS488 devices should ignore this message. The second byte is a GPIB secondary command that indicates the number of meters of cable in your system. It is called the Configure (CFGn) message. Because HS488 can operate only with cable lengths of 1 to 15 meters, only CFGn values of 1 through 15 (hex 61 through 6F) are valid. If the cable length was configured correctly in `ibconf`, you can determine how many meters of cable are in your system by calling `ibask` (option `IbaHSCableLength`) in your application program. For CFE and CFGn messages, refer to Appendix A, *Multiline Interface Messages*, in the *NI-488.2 Function Reference Manual for DOS/Windows*.

### System Configuration Effects on HS488

Maximum data transfer rates can be limited by your host computer and GPIB system setup. For example, even though the theoretical maximum transfer rate with HS488 is 8 Mbytes/s, the maximum transfer rate obtainable on PC-compatible computers with an ISA bus is 2 Mbytes/s. The same IEEE 488 cabling constraints for a 350 ns T1 delay apply to HS488. As you increase the amount of cable in your GPIB configuration, the maximum data transfer rate using HS488 decreases. For example, two HS488 devices connected by two meters of cable can transfer data faster than three HS488 devices connected by four meters of cable.

## Waiting for GPIB Conditions

You can use the `ibwait` function to obtain the current `ibsta` value or to suspend your application until a specified condition occurs on the GPIB. If you use `ibwait` with a parameter of zero, it immediately updates `ibsta` and returns. If you want to use `ibwait` to wait for one or more events to occur, then pass a wait mask to the function. The wait mask should always include the TIMO event; otherwise, your application is suspended indefinitely until one of the wait mask events occurs.

## Device-Level Calls and Bus Management

The NI-488 device-level calls are designed to perform all of the GPIB management for your application program. However, the NI-488.2 driver can handle bus management only when the GPIB interface board is CIC (Controller-In-Charge). Only the CIC is able to send command bytes to the devices on the bus to perform device addressing or other bus management activities. Use one of the following methods to make your GPIB board the CIC:

- If your GPIB board is configured as the System Controller (default), it automatically makes itself the CIC by asserting the IFC line the first time you make a device-level call.
- If your setup includes more than one Controller, or if your GPIB interface board is not configured as the System Controller, use the CIC Protocol method. To use the protocol, issue the `ibconfig` function (option `IbcCICPROT`) or use the `ibconf` configuration utility to activate the CIC protocol. If the interface board is not CIC and you make a device-level call with the CIC Protocol enabled, the following sequence occurs:

  1. The GPIB interface board asserts the SRQ line.
  2. The current CIC serial polls the board.
  3. The interface board returns a response byte of hex 42.
  4. The current CIC passes control to the GPIB board.

If the current CIC does not pass control, the NI-488.2 driver returns the ECIC error code to your application. This error can occur if the current CIC does not understand the CIC Protocol. If this happens, you could send a device-specific command requesting control for the GPIB board. Then use a board-level `ibwait` command to wait for CIC.

# Talker/Listener Applications

Although designed for Controller-In-Charge applications, you can also use the NI-488.2 software in most non-Controller situations. These situations are known as Talker/Listener applications because the interface board is not the GPIB Controller. A typical Talker/Listener application waits for events from the Controller and responds as appropriate. The following paragraphs describe some programming techniques for Talker/Listener applications.

## Waiting for Messages from the Controller

A Talker/Listener application typically uses `ibwait` with a mask of 0 to monitor the status of the interface board. Then, based on the status bits set in `ibsta`, the application takes whatever action is appropriate. For example, the application could monitor the status bits TACS (Talker Active State) and LACS (Listener Active State) to determine when to send data to or receive data from the Controller. The application could also monitor the DCAS (Device Clear Active State) and DTAS (Device Trigger Active State) bits to determine if the Controller has sent the device clear (DCL or SDC) or trigger (GET) messages to the interface board. If the application detects a device clear from the Controller, it might reset the internal state of message buffers. If it detects a trigger message from the Controller, the application might begin an operation such as taking a voltage reading if the application is actually acting as a voltmeter.

## Using the Event Queue

Some applications need to know the order in which certain messages are sent by the Controller. To monitor the ordering of these messages, your application program must enable the EVENT bit, using `ibconfig` (option `IbcEventQueue`). When the EVENT bit is enabled, the DCAS and DTAS bits are no longer active. Instead, all DCAS and DTAS messages are stored in a queue, in the order that they are received. The event queue also stores interface clear (IFC) messages. When the queue contains some information, the NI-488.2 software sets the EVENT bit in `ibsta`. When the application program detects EVENT, it can call the function `ibevent` to retrieve the first event that occurred. Retrieving events from the queue ensures that the application can respond to device clear, device trigger, and interface clear messages in the correct order.

### Requesting Service

Another type of event that might be important in a Talker/Listener application is the serial poll. A Talker/Listener application can call `ibrsv` with a serial poll response byte when it needs to request service from the Controller. If the application needs to know when the Controller has read the serial poll response byte, it can enable the SPOLL bit in `ibsta` using `ibconfig`, option `IbcSPollBit`. The NI-488.2 software sets the SPOLL bit when the Controller serial polls the board.

### Simulating Multiple Addresses

With the NI-488.2 software, Talker/Listener applications can *virtualize* multiple GPIB addresses. By virtualizing several GPIB addresses, the Talker/Listener application appears as multiple, distinct GPIB devices to the Controller. You might use this feature to write an application that behaves differently depending on which GPIB address is sent to it by the Controller.

Using the `GotoMultAddr` function, an application registers two routines with the NI-488.2 driver. The driver calls the first routine whenever the Controller sends a GPIB address on the bus. The routine must decide to accept or reject this address. If it accepts the address then the interface board responds to the new address for subsequent operations. When the Controller sends another address, the process is repeated. The second routine is called by the driver when the Controller serial polls one of the virtualized addresses. This routine must provide the appropriate serial poll response byte, which the driver sends to the Controller.

## Serial Polling

You can use serial polling to obtain specific information from GPIB devices when they request service. When the GPIB SRQ line is asserted, it signals the Controller that a service request is pending. The Controller must then determine which device asserted the SRQ line and respond accordingly. The most common method for SRQ detection and servicing is the serial poll. This section describes how you can set up your application to detect and respond to service requests from GPIB devices.

### Service Requests from IEEE 488 Devices

IEEE 488 devices request service from the GPIB Controller by asserting the GPIB SRQ line. When the Controller acknowledges the SRQ, it serial polls each open device on the bus to determine which device requested service. Any device requesting service returns a status byte with bit 6 set and then unasserts the SRQ line. Devices not requesting service return a status byte with bit 6 cleared. Manufacturers of IEEE 488 devices use lower order bits to communicate the reason for the service request or to summarize the state of the device.

## Service Requests from IEEE 488.2 Devices

The IEEE 488.2 standard refined the bit assignments in the status byte. In addition to setting bit 6 when requesting service, IEEE 488.2 devices also use two other bits to specify their status. Bit 4, the Message Available bit (MAV), is set when the device is ready to send previously queried data. Bit 5, the Event Status bit (ESB), is set if one or more of the enabled IEEE 488.2 events occurs. These events include power-on, user request, command error, execution error, device-dependent error, query error, request control, and operation complete. The device can assert SRQ when ESB or MAV are set, or when a manufacturer-defined condition occurs.

## Automatic Serial Polling

You can enable automatic serial polling if you want your application to conduct a serial poll automatically any time the SRQ line is asserted. You can use autopolling with NI-488 device-level calls only. The autopolling procedure occurs as follows:

1. To enable autopolling, use the configuration utility, `ibconf`, or the configuration function, `ibconfig` with option `IbcAUTOPOLL`. (Autopolling is enabled by default.)

2. When the SRQ line is asserted, the driver automatically serial polls the open devices.

3. Each positive serial poll response (bit 6 or hex 40 is set) is stored in a queue associated with the device that sent it. The RQS bit of the device status word, `ibsta`, is set.

4. The polling continues until SRQ is unasserted or an error condition is detected.

5. To empty the queue, use the `ibrsp` function. `ibrsp` returns the first queued response. 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, a serial poll is conducted and returns whatever response is received. You should empty the queue as soon as an automatic serial poll occurs, because responses might be discarded if the queue is full.

6. If the RQS bit of the status word is still set after `ibrsp` is called, the response byte queue contains at least one more response byte. If this happens, you should continue to call `ibrsp` until RQS is cleared.

### Stuck SRQ State

If autopolling is enabled and the GPIB interface board detects an SRQ, the driver serial polls all open devices connected to that board. The serial poll continues until either SRQ unasserts or all the devices have been polled.

If no device responds positively to the serial poll, or if SRQ remains in effect because of a faulty instrument or cable, a *stuck SRQ* state is in effect. If this happens during an `ibwait` for RQS, the driver reports the ESRQ error. If the *stuck SRQ* state happens, no further polls are attempted until another `ibwait` for RQS is made. Whenever `ibwait` is issued, the *stuck SRQ* state is terminated and the driver attempts a new set of serial polls.

### Autopolling and Interrupts

If autopolling and interrupts are both enabled, the NI-488.2 software can perform autopolling after any device-level NI-488 call as long as no GPIB I/O is currently in progress. This means that an automatic serial poll can occur even when your application is not making any calls to the NI-488.2 software. Autopolling can also occur when a device-level `ibwait` for RQS is in progress. Autopolling is not allowed whenever an application calls a board-level NI-488 function or any NI-488.2 routine, or the *stuck SRQ* (ESRQ) condition occurs.

If autopolling is enabled and interrupts are disabled, you can use autopolling in the following situations only:

- During a device-level `ibwait` for RQS
- Immediately after a device-level NI-488 function is completed, before control is returned to the application program.

### "ON SRQ" Functionality

With the NI-488.2 software, applications can respond asynchronously to the assertion of the GPIB SRQ line. When an application has enabled the *ON SRQ* feature of the software, a user-supplied function is called whenever the GPIB SRQ line is asserted. This makes it possible to customize an application's response when a device requests service.

**Note:** ***Automatic serial polling must be disabled (*`ibconfig`*, option* `IbcAUTOPOLL`*) for the ON SRQ feature to work properly***.

### C "ON SRQ" Capability

C applications can respond asynchronously to SRQ using the NI-488 `ibsrq` function. This function lets an application specify an SRQ-handling routine that is called whenever the NI-488.2 driver detects that the SRQ line is asserted. This SRQ-handling routine is *not* an interrupt service routine. The driver checks the GPIB SRQ line after any NI-488 function or NI-488.2 routine has completed, and if SRQ is asserted and the application has called `ibsrq`, the user-defined SRQ-handling routine is called.

### BASIC/QuickBASIC/BASICA "ON SRQ" Capability

BASIC/QuickBASIC/BASICA applications can also respond asynchronously to SRQ. The BASIC languages provide an *ON PEN* capability which the NI-488.2 driver automatically converts to an ON SRQ capability. Your BASIC application can enable the ON SRQ functionality by executing the following sample lines of code:

```
100  REM Define srq-handling routine (MySRQRoutine)
200  ON PEN GOSUB MySRQRoutine
300  REM Enable the ON SRQ functionality
400  PEN ON
```

Your BASIC application must define the `MySRQRoutine` SRQ-handling routine. This routine is called whenever the assertion of the SRQ line is detected by the BASIC environment. This SRQ-handling routine is *not* an interrupt service routine. The BASIC environment only responds to SRQ in between executing lines of BASIC code.

## SRQ and Serial Polling with NI-488 Device Functions

You can use the device-level NI-488 function `ibrsp` to conduct a serial poll. `ibrsp` conducts a single serial poll and returns the serial poll response byte to the application program. If automatic serial polling is enabled, the application program can use `ibwait` to suspend program execution until RQS appears in the status word, `ibsta`. The program can then call `ibrsp` to obtain the serial poll response byte.

The following example illustrates the use of the `ibwait` and `ibrsp` functions in a typical SRQ servicing situation when automatic serial polling is enabled.

```
#include "decl.h"

char GetSerialPollResponse ( int DeviceHandle )
{
       char SerialPollResponse = 0;

       ibwait ( DeviceHandle, TIMO | RQS );

       if ( ibsta & RQS )  {
              printf ( "Device asserted SRQ.\n" );
              /*  Use ibrsp to retrieve the serial poll
                  response.  */
              ibrsp ( DeviceHandle, &SerialPollResponse );
       }
       return SerialPollResponse;
}
```

## SRQ and Serial Polling with NI-488.2 Routines

The NI-488.2 software includes a set of NI-488.2 routines that you can use to conduct SRQ servicing and serial polling. Routines pertinent to SRQ servicing and serial polling are `AllSpoll`, `FindRQS`, `ReadStatusByte`, `TestSRQ`, and `WaitSRQ`.

`AllSpoll` can serial poll multiple devices with a single call. It places the status bytes from each polled instrument into a predefined array. Then you must check the RQS bit of each status byte to determine whether that device requested service.

`ReadStatusByte` is similar to `AllSpoll`, except that it serial polls only a single device. It is also analogous to the device-level NI-488 `ibrsp` function.

`FindRQS` serial polls a list of devices until it finds a device that is requesting service or until it has polled all of the specified devices. The routine returns the index and status byte value of the device requesting service.

`TestSRQ` determines whether the SRQ line is asserted or unasserted, and returns to the program immediately.

`WaitSRQ` is similar to `TestSRQ`, except that `WaitSRQ` suspends the application program until either SRQ is asserted or the timeout period is exceeded.

The following examples use NI-488.2 routines to detect SRQ and then determine which device requested service. In these examples three devices are present on the GPIB at addresses 3, 4, and 5, and the GPIB interface is designated as bus index 0. The first example uses `FindRQS` to determine which device is requesting service and the second example uses `AllSpoll` to serial poll all three devices. Both examples use `WaitSRQ` to wait for the GPIB SRQ line to be asserted.

**Note**: ***Automatic serial polling is not used in these examples because you cannot use it with NI-488.2 routines***.

### Example 1: Using FindRQS

This example illustrates the use of `FindRQS` to determine which device is requesting service.

```
void GetASerialPollResponse ( char *DevicePad,
                              char *DeviceResponse )
{
    char SerialPollResponse = 0;
    int WaitResult;
    Addr4882_t Addrlist[4] = {3,4,5,NOADDR};

    WaitSRQ (0, &WaitResult);

    if (WaitResult) {
       printf ("SRQ is asserted.\n");

/* Use FindRQS to find a device that requested service. */

     FindRQS ( 0, AddrList, &SerialPollResponse );
     if (!(ibsta & ERR))  {
        printf ("Device at pad %x returned byte %x.\n",
                 AddrList[ibcnt],(int) SerialPollResponse);
        *DevicePad = AddrList[ibcnt];
        *DeviceResponse = SerialPollResponse;
     }
   }

   return;
}
```

**Example 2: Using AllSpoll**

This example illustrates the use of `AllSpoll` to serial poll three devices.

```
void GetAllSerialPollResponses ( Addr4882_t AddrList[], short
ResponseList[] )
{
   int WaitResult;

   WaitSRQ (0, &WaitResult);

   if ( WaitResult ) {
      printf ( "SRQ is asserted.\n" );

/* Use Allspoll to serial poll all the devices at once. */

      AllSpoll ( 0, AddrList, ResponseList );
      if (!(ibsta & ERR))  {
         for ( i = 0; AddrList[i] != NOADDR; i++ )  {
            printf ("Device at pad %x returned byte %x.\n",
                    AddrList[i], ResponseList[i] );
         }
      }
   }

   return;
}
```

# Parallel Polling

Although parallel polling is not widely used, it is a useful method for obtaining the status of more than one device at the same time. The advantage of a parallel poll is that it can easily check up to eight individual devices at once. In comparison, eight separate serial polls would be required to check eight devices for their serial poll response bytes.

## Implementing a Parallel Poll

You can implement parallel polling with either NI-488 functions or NI-488.2 routines. If you use NI-488.2 routines to execute parallel polls, you do not need extensive knowledge of the parallel polling messages. However, you should use the NI-488 functions for parallel polling when the GPIB board is not the Controller and must configure itself for a parallel poll and set its own individual status bit (`ist`).

**Parallel Polling with NI-488 Functions**

Follow these steps to implement parallel polling using NI-488 functions. Each step contains example code.

1. Configure the device for parallel polling using the `ibppc` function, unless the device can configure itself for parallel polling.

   `ibppc` requires an 8-bit value to designate the data line number, the `ist` sense, and whether or not the function configures or unconfigures the device for the parallel poll. The bit pattern is as follows:

   0 1 1 E S D2 D1 D0

   E is 1 to disable parallel polling and 0 to enable parallel polling for that particular device.

   S is 1 if the device is to assert the assigned data line when `ist` = 1, and 0 if the device is to assert the assigned data line when `ist` = 0.

   D2 through D0 determine the number of the assigned data line. The physical line number is the binary line number plus one. For example, DIO3 has a binary bit pattern of 010.

   The following example code configures a device for parallel polling using NI-488 functions. The device asserts DIO7 if its `ist` = 0.

   In this example, the `ibdev` command is used to open a device that has a primary address of 3, has no secondary address, has a timeout of 3 s, asserts EOI with the last byte of a write operation, and has EOS characters disabled.

   ```
   #include "decl.h"
   char ppr;

   dev = ibdev(0,3,0,T3s,1,0);

   /* Pass the binary bit pattern, 0110 0110 or hex 66, to
      ibppc. */

   ibppc(dev, 0x66);
   ```

   If the GPIB interface board configures itself for a parallel poll, you should still use the `ibppc` function. Pass the board index or a board unit descriptor value as the first argument in `ibppc`. In addition, if the individual status bit (`ist`) of the board needs to be changed, use the `ibist` function.

   In the following example, the GPIB board is to configure itself to participate in a parallel poll. It asserts DIO5 when `ist` = 1 if a parallel poll is conducted.

   ```
   ibppc(0, 0x6C);
   ibist(0, 1);
   ```

2. Conduct the parallel poll using `ibrpp` and check the response for a certain value. The following example code performs the parallel poll and compares the response to hex 10, which corresponds to DIO5. If that bit is set, the `ist` of the device is 1.

```
ibrpp(dev, &ppr);
if (ppr & 0x10) printf("ist = 1\n");
```

3. Unconfigure the device for parallel polling with `ibppc`. Notice that any value having the parallel poll disable bit set (bit 4) in the bit pattern disables the configuration, so you can use any value between hex 70 and 7E.

```
ibppc(dev, 0x70);
```

**Parallel Polling with NI-488.2 Routines**

Follow these steps to implement parallel polling using NI-488.2 routines. Each step contains example code.

1. Configure the device for parallel polling using the `PPollConfig` routine, unless the device can configure itself for parallel polling. The following example configures a device at address 3 to assert data line 5 (DIO5) when its `ist` value is 1.

```
#include "decl.h"
char response;
Addr4882_t AddressList[2];

/* The following command clears the GPIB.   */

SendIFC(0);

/* The value of sense is compared with the ist bit of the
   device and determines whether the data line is asserted. */

PPollConfig(0,3,5,1);
```

2. Conduct the parallel poll using `PPoll`, store the response, and check the response for a certain value. In the following example, because DIO5 is asserted by the device if `ist` = 1, the program checks bit 4 (hex 10) in the response to determine the value of `ist`.

```
PPoll(0, &response);

/* If response has bit 4 (hex 10) set, the ist bit of the
   device at that time is equal to 1.  If it does not appear,
   the ist bit is equal to 0.  Check the bit in the following
   statement. */

if (response & 0x10) {
   printf("The ist equals 1.\n");
}
else {
   printf("The ist equals 0.\n");
}
```

3. Unconfigure the device for parallel polling using the `PPollUnconfig` routine as shown in the following example. In this example, the `NOADDR` constant must appear at the end of the array to signal the end of the address list. If `NOADDR` is the only value in the array, all devices receive the parallel poll disable message.

```
AddressList[0] = 3;
AddressList[1] = NOADDR;
PPollUnconfig(0, AddressList);
```

# Chapter 8
# ibconf–Interface Bus Configuration Utility

This chapter contains a description of `ibconf`, the software configuration program you can use to configure the NI-488.2 software.

## Overview

The `ibconf` utility is a screen-oriented, interactive program you can use to view or modify the configuration parameters of your GPIB interface boards and the GPIB devices connected to them. Usually, you use `ibconf` to modify the hardware settings of your GPIB interface boards.

The `ibconf` utility can read in and display configuration parameters for the NI-488.2 driver file on disk. You can then save the changes back to the disk file.

Instead of using `ibconf`, you can configure your driver programmatically by using the `ibconfig` function to alter any of the board or device characteristics while your program is running. If you use dynamic configuration, you do not need to run `ibconf` before you start your application. Also, you can run your application on any computer with the appropriate NI-488.2 software regardless of its configuration because the application configures the driver as necessary. Programmatic configuration is the preferred method of GPIB application development.

## Starting ibconf

The simplest way to use `ibconf` is to change to the directory that contains the installed NI-488.2 software and enter the following command:

```
ibconf
```

`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 8-1 lists the options you can select when you start `ibconf`.

Table 8-1. Options for Starting ibconf

| **`ibconf` Option** | **Action** |
|---|---|
| *`driver`* | Configure a specific driver.<br>`ibconf` configures the given driver file instead of searching for a `gpib.com` file (for example, `ibconf d:\at-gpib\at-gpib.com`). |
| `-h` | Help.<br>This option causes `ibconf` to display a summary of options. |
| `-d` | Dynamic configuration.<br>This option causes `ibconf` to automatically update the driver loaded in memory when you exit `ibconf`. See the *Exiting ibconf* section later in this chapter. |
| `-e` | Expert mode.<br>This option disables `ibconf` warning messages when you exit `ibconf`. See the *Exiting ibconf* section later in this chapter. |
| `-f` | Disable dynamic configuration.<br>This option causes `ibconf` to ignore the driver loaded in memory when you exit `ibconf`. `ibconf` does not try to update the loaded driver. |
| `-m` | Monochrome mode.<br>This option causes `ibconf` to run in monochrome mode even if you have a color monitor. |
| `-vb` | Video access through BIOS.<br>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 systems. |

# Upper and Lower Levels of ibconf

`ibconf` operates at both an upper and a lower level. The upper level consists of the device map, which gives a graphical picture of the GPIB system. The lower level consists of screens describing the individual boards and devices that make up the system.

## Upper-Level Device Map

Figure 8-1 shows the upper level of `ibconf`.

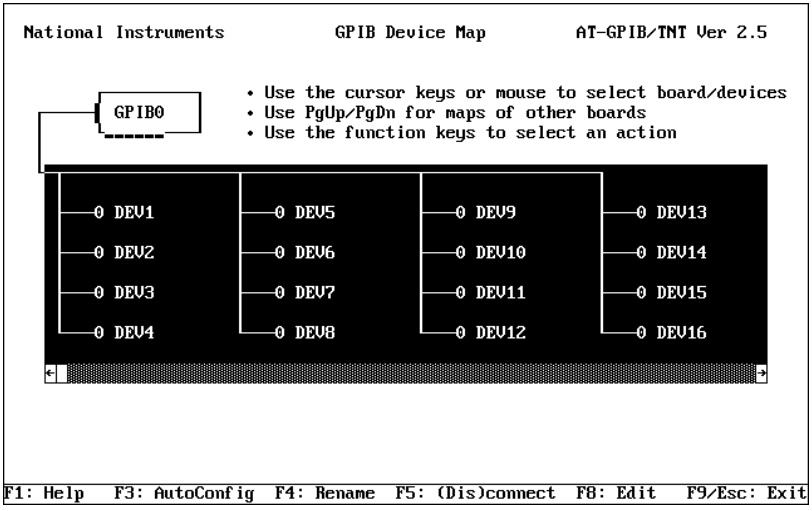



Figure 8-1. Upper Level of ibconf

As shown in Figure 8-1, 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 each interface board. You can move around the map by using the cursor control keys or the mouse.

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.

**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 want 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 in lowercase or uppercase.

The following restrictions apply when renaming a device:

- Extensions (.xxx) are not allowed
- As specified by DOS, the device name cannot use the following characters:

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

- Do not use the reserved names `con` or `nul` for your device
- Do not give GPIB device names the same names as files, directories, 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.
- The access board names `gpib0`, `gpib1`, `gpib2`, and `gpib3` are fixed and cannot be changed.

**(Dis)connect**

Use the function key <F5> to connect or disconnect a device from a particular access board. Move the cursor to the device that you want to connect or disconnect by using the cursor control keys and then 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 then 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, press the function key <F9> or <Escape>.

**Output GPIB Driver Configuration**

When configuring a GPIB driver, a text version of the driver configuration can be written to a disk file. Use the function key <F2> to cause `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> for automatic configuration. When autoconfiguring a particular GPIB board, `ibconf` finds all of the listening devices and adjusts the device map for the board so that only those 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 takes only a few seconds. Make sure that all the devices in the system are connected and powered on before running autoconfigure.

**Caution:** ***Once you press <F3> to autoconfigure, you cannot undo the new configuration.***

If you are using more than one interface board in your system and you want to use the autoconfigure feature, start with the lowest-numbered board and autoconfigure additional boards in ascending order. You should use ascending order (`gpib0`, then `gpib1`, and so on) because `ibconf` might disconnect devices from higher-numbered interface boards when autoconfiguring lower-numbered boards.

**Exit**

Use the function key <F9> or <Escape> to exit `ibconf`. If you have made changes, `ibconf` prompts you to save the changes before exiting. For more information, refer to the *Exiting ibconf* section later in this chapter.

## Lower-Level Device/Board Characteristics

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 8-2.

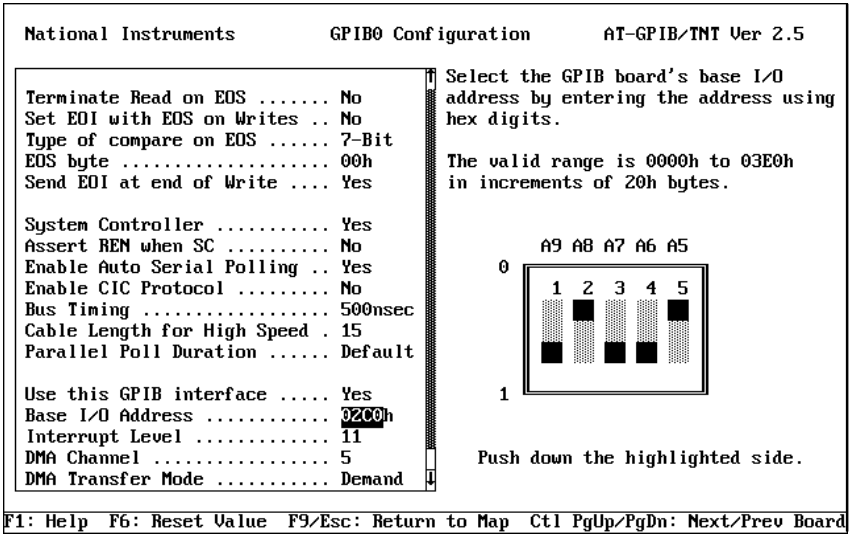



Figure 8-2. Lower Level of ibconf

You can access the lower-level screens from the map level of `ibconf` by selecting a board or device and pressing <F8>. You can use the <Up>, <Down>, <PageUp>, and <PageDown> cursor keys to select a characteristic. For your convenience, cursor control keys and function keys are defined at the bottom of your computer screen.

Selecting the configuration settings for each device and board customizes the communications and other options to be used with that board or device. The access board uses these settings either when device functions are called to program the device or when board functions are used to program the board.

The following options 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, the field for that characteristic. You can also use <PageUp>, <PageDown>, <Home>, or <End> to select other characteristics of a device or 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 press <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`.

# Board and Device Configuration Options

To view detailed information about each characteristic, position the cursor in the field for that characteristic. For information on characteristics specific to a particular driver, check the getting started manual that came with your interface board. The following paragraphs describe the options available in `ibconf` for the NI-488.2 software for DOS.

## Primary GPIB Address

All devices and boards must be assigned a unique primary address in the range hex 00 to hex 1E (0 to 30 decimal). The default primary address of all GPIB boards is 0.

The GPIB primary address of any device is set within the device, either with hardware switches or by a software program. The address set within the device must correspond to the address in the memory-resident driver. In the NI-488.2 driver for DOS, the default primary addresses of `dev1` through `dev16`, and `dev17` through `dev32`, are 1 through

32, respectively. Refer to the device documentation for instructions about setting the device address. GPIB boards do *not* have hardware switches to select the GPIB address.

The primary GPIB address is used to compute the talk and listen addresses of devices and boards. The NI-488.2 driver automatically forms a listen address by adding hex 20 to the primary address and a talk address by adding hex 40 to the primary address. For example, a primary address of hex 10 has a listen address of hex 30 and a talk address of hex 50.

## Secondary GPIB Address

Any device or board using extended addressing must be assigned a secondary address in the range hex 60 to hex 7E (96 to 126 decimal), or you can select `NONE` 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 by a software program. The address set within the device must correspond to the address in the memory-resident driver. Refer to the device documentation for instructions about secondary addressing. The default setting for this characteristic is `NONE` for all boards and devices.

## Timeout Setting

The timeout value is the approximate length of time that GPIB functions wait for data to be transferred or commands to be sent. 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. For example, if the SRQI bit and TIMO bit in the event mask are passed to the `ibwait` function and no SRQ is detected, the function times out after the timeout period elapses. The default option for this characteristic is 10 s.

## 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 488.1 specification does not specify the length of time a Controller should wait for the response byte. The default of 1 s 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 in 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 Writes

A `yes` response in 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 can choose whether to compare all eight bits or just the seven least significant bits, which are in ASCII or ISO (International Standards Organization) format. The default option for this characteristic is `7-bit`.

**Note**: ***This field is only meaningful if a*** `yes` ***response was given for either the*** `Set EOI with EOS on Write` ***field or the*** `Terminate Read on EOS` ***field.***

## EOS Byte

Some devices can be programmed to terminate a read operation when a selected character is detected. A linefeed character (hex 0A) is a common EOS byte. The default option for this characteristic is 00H.

**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 EOS byte is used only so that the driver terminates read operations properly.***

## Send 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 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. In some situations, such as a network of computers linked by the GPIB, another device may be System Controller. In these situations, you should designate the GPIB board as *not* System Controller by selecting `no` for this field. A `yes` response would designate System Controller capability. The GPIB board is usually designated as System Controller. The default option for this characteristic is `yes`.

**Note**: ***You should not have more than one designated System Controller in any GPIB system.***

## 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, if that the board has System Controller capability. If you give a `no` response, 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. Normally, this feature does not conflict with devices that conform to the IEEE 488.1 specification. If a conflict exists with a device, use a `no` response for this field to disable this feature. The default option for this characteristic is `yes`.

## Enable CIC Protocol (Board Characteristic Only)

If a device-level NI-488 function is called 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 wants 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, or if the CIC protocol is disabled, the ECIC error is returned. The default option for this characteristic is `no`.

## 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, the value of 350 ns is appropriate.

Other factors might affect the choice of the T1 delay, although they are unlikely to affect your system setup. Refer to the ANSI/IEEE Standard 488.1-1987, Section 5.2, for more information about these other factors. The default for this option is 500 ns.

## Cable Length for High Speed (Board Characteristic Only)

This field specifies the number of meters of GPIB cable you have in your system. If you use the HS488 high-speed protocol to communicate with HS488-compliant devices, you must specify the total number of meters of GPIB cable in your system. The System Controller, when it initializes the GPIB, must send this information to all HS488 devices so high-speed transfers occur without errors.

## 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) use the default duration of 2 µs. If you are using a GPIB bus extender in transparent parallel poll mode, you should increase the poll duration to 10 µs so that the bus extender can operate transparently to your applications.

## Use This GPIB Interface (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 returns the EDVR error as soon as a program tries to access the board. By default, access board `gpib0` is enabled, and `gpib1`, `gpib2`, and `gpib3` are disabled.

## Base I/O Address (Board Description 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.

**Note**: ***On some systems, this field is read-only and you cannot change it using*** `ibconf`***. Refer to your getting started manual for information on how to change the I/O address.***

## DMA Channel (Board Characteristic Only)

This field specifies the DMA channel used by the GPIB board. It must be set to the same value as the DMA channel (arbitration level for Micro Channel systems) setting for the GPIB board itself. Setting the DMA channel is explained in the getting started manual that you received with your GPIB interface board.

**Note**: ***On some systems, you can only enable or disable the use of DMA with the*** `ibconf` ***utility. You cannot change the DMA channel setting. Refer to your getting started manual for information on how to change the DMA channel.***

## Interrupt Jumper Setting (Board Characteristic Only)

This field specifies the interrupt line used by the GPIB 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.

**Note**: ***On some systems, you can only enable or disable the use of interrupts with the*** `ibconf` ***utility. You cannot change the IRQ request line. Refer to your getting started manual for information on how to change the interrupt level.***

### Enable Repeat Addressing (Device Characteristic Only)

Normally, devices are not 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. Avoiding readdressing saves some time when you have several GPIB operations to perform. But it might be a problem for some older IEEE 488.1 devices that require their GPIB address to be sent with each I/O operation. Select `yes` to enable repeat addressing in such a situation. The default option for this characteristic is `no`.

### 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. Use this field to indicate the type of board that is installed in your system. You can have both a GPIB-PCII and a GPIB-PCIIA interface board installed in your system at the same time.

## Default Configurations in ibconf

This section lists the default configuration values of the NI-488.2 driver.

- Thirty-two devices with symbolic names `dev1` through `dev32`.
- Four access boards with symbolic names `gpib0`, `gpib1`, `gpib2`, and `gpib3`. You cannot change the access board names.
- 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, `dev` is at address 1. These 16 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 for 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 timeout value on I/O and wait function calls is set for 10 s.
- Each GPIB board and device is set to perform I/O transfers using DMA.
- Automatic serial polling is enabled.
- At the end of each NI-488.2 routine, the NI-488.2 driver leaves the bus in its currently addressed state (IEEE 488.2 standard).

# Exiting ibconf

After you have made all your changes, you can exit `ibconf` by pressing <F9> or <Esc>. The program prompts you to save the changes. Select `yes` to save the changes, `no` to discard the changes, or `cancel` to remain in `ibconf`. If you select `yes`, `ibconf` asks if you want the changes take effect immediately. Select `yes` or `no`. If you choose `no` or `ibconf` is unable to make the changes take effect immediately, `ibconf` displays a final message that instructs you to restart your computer.

## Checking for Errors

Unless you started in expert mode, `ibconf` checks for possible problems before quitting. It alerts you to situations such as the following:

- GPIB addressing conflict between a device and its access board
- GPIB boards not present in the host machine at the specified address
- Timeouts disabled on a device or board

To disable automatic checking when starting `ibconf`, use the `-e` option.

# Appendix A
# Status Word Conditions

This appendix gives a detailed description of the conditions reported in the status word, `ibsta`.

For information about how to use `ibsta` in your application program, refer to Chapter 3, *Developing Your Application*.

If a function call returns an ENEB or EDVR error, all status word bits except the ERR bit are cleared, indicating that it is not possible to obtain the status of the GPIB board.

Each bit in `ibsta` can be set for device calls (dev), board calls (brd), or both (dev, brd).

The following table lists the status word bits.

Table A-1. Status Word Bits

| Mnemonic | Bit Pos. | Hex Value | 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 |
| SPOLL | 10 | 400 | brd | Board has been serial polled by Controller |
| EVENT | 9 | 200 | brd | DCAS, DTAS, or IFC event has occurred |
| 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 |

## ERR (dev, brd)

ERR is set in the status word following any call that results in an error. You can determine the particular error by examining the error variable `iberr`. Appendix B, *Error Codes and Solutions*, describes error codes that are recorded in `iberr` along with possible solutions. ERR is cleared following any call that does not result in an error.

## TIMO (dev, brd)

TIMO indicates that the timeout period has been exceeded. TIMO is set in the status word following an `ibwait` call if the TIMO bit of the `ibwait` mask parameter is set and the time limit expires. TIMO is also set following any synchronous I/O functions (for example, `ibcmd`, `ibrd`, `ibwrt`, `Receive`, `Send`, and `SendCmds`) if a timeout occurs during one of these calls. TIMO is cleared in all other circumstances.

## END (dev, brd)

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

Some applications might 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 (option `IbcEndBitIsNormal`) to enable a mode in which the END bit is set only when EOI is asserted. In this mode, if the I/O operation completes because of the EOS character by itself, END is not set. The application should check the last byte of the received buffer to see if it is the EOS character.

## SRQI (brd)

SRQI indicates that a GPIB device is requesting service. SRQI is set whenever the GPIB board is CIC, the GPIB SRQ line is asserted, and the automatic serial poll capability is disabled. SRQI is cleared either when the GPIB board ceases to be the CIC or when the GPIB SRQ line is unasserted.

## RQS (dev)

RQS appears in the status word only after a device-level call. It indicates that one or more serial poll response bytes are waiting in the device's serial poll response queue. Automatic serial poll responses are not stored in the response queue unless they have bit 6 set.

An automatic serial poll occurs either as a result of a call to `ibwait`, or automatically, if automatic serial polling is enabled. If the serial poll response queue is not empty, `ibrsp` returns the oldest byte stored in the queue. To empty the response queue, call `ibrsp` repeatedly until RQS is no longer set in the device's status word.

## SPOLL (brd)

Use SPOLL in Talker/Listener applications to determine when the Controller has serial polled the GPIB board. The SPOLL bit is disabled by default. Use the `ibconfig` function (option `IbcSPollBit`) to enable it. When this bit is enabled, it is set after the board has been polled. SPOLL is cleared 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`.

## EVENT (brd)

Use EVENT in Talker/Listener applications (applications in which the GPIB interface is not the Controller) to monitor the order of GPIB device clear, group execute trigger, and send interface clear commands. The usual DCAS and DTAS bits of `ibsta` might be insufficient.

The EVENT bit is disabled by default. If you want to use this bit, you must use the `ibconfig` function (option `IbcEventQueue`) to enable this bit. When you enable this bit, 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 can then call the `ibevent` function to determine which event occurred.

## CMPL (dev, brd)

CMPL indicates the condition of I/O operations. It is set whenever an I/O operation is complete. CMPL is cleared while the I/O operation 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 that the Local Lockout (LLO) message has been sent either by the GPIB board or by another Controller. LOK is cleared when the System Controller unasserts the Remote Enable (REN) GPIB line.

## 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 in the following situations:

- 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
- 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 either while the GPIB board is System Controller or when another Controller passes control to the GPIB board. CIC is cleared either when 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 it is cleared when the ATN line is unasserted.

## TACS (brd)

TACS indicates whether the GPIB board is addressed as a Talker. TACS is set whenever the GPIB board detects that 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 is addressed as a Listener. LACS is set whenever the GPIB board detects that 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 that the `ibgts` function has been 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 that the Group Execute Trigger (GET) command has been sent by another Controller. DTAS is cleared 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 that the Device Clear (DCL) command has been sent by another Controller, or whenever the GPIB board as a Listener detects that the Selected Device Clear (SDC) command has been sent by another Controller. DCAS is cleared on any call immediately following an `ibwait` call, if the DCAS bit was set in the `ibwait` mask parameter. It also clears on any call immediately following a read or write.

# Appendix B
# Error Codes and Solutions

This appendix lists a description of each error, some conditions under which it might occur, and possible solutions.

The following table lists the GPIB error codes.

Table B-1. GPIB Error Codes

| Error Mnemonic | iberr Value | Meaning |
|---|---|---|
| EDVR | 0 | DOS error |
| ECIC | 1 | Function requires GPIB board to be CIC |
| ENOL | 2 | No Listeners on the GPIB |
| 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 |
| 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 |

## EDVR (0)

EDVR is returned when the board or device name passed to `ibfind` is not configured in the software. In this case, the variable `ibcntl` contains the DOS error code 2, *Device Not Found* or 110, *Open failed.*

EDVR is also returned when an invalid unit descriptor is passed to any function call. In this case, the variable `ibcntl` contains the DOS error code 6, *Invalid handle*.

EDVR is also returned when the driver (`gpib.com`) is not installed.

### Solutions

- Use `ibdev` to open a device without specifying its symbolic name.
- Use only device or board names that are configured in the utility program `ibconf` as parameters in the `ibfind` function.
- Use the unit descriptor returned from the `ibfind` function as the first parameter in subsequent NI-488 functions. Examine the variable after the `ibfind` and before the failing function to make sure it was not corrupted.
- Make sure the NI-488.2 driver is installed by checking the `config.sys` file in the root directory. Make sure it contains the following line:

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

  where *`dir`* is the directory that contains `gpib.com`.

## ECIC (1)

ECIC is returned when one of the following board functions or routines is called while the board is not CIC:

- Any device-level NI-488 functions that affect the GPIB
- Any board-level NI-488 functions that issue GPIB command bytes such as `ibcmd`, `ibcmda`, `ibln`, `ibrpp`
- `ibcac`, `ibgts`
- Any of the NI-488.2 routines that issue GPIB command bytes such as `SendCmds`, `PPoll`, `Send`, `Receive`

### Solutions

- Use `ibsic` or `SendIFC` to make the GPIB board become CIC on the GPIB.
- Use `ibrsc 1` to make sure your GPIB board is configured as System Controller.
- In multiple CIC situations, always be certain that the CIC bit appears in the status word ibsta before attempting these calls. If it does not appear, you can perform an `ibwait` (for 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.

ENOL can 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.

## Solutions

- Make sure that the GPIB address of your device matches the GPIB address of the device to which you want to write data.
- Use the appropriate hex code in `ibcmd` to address your device.
- Check your cable connections and make sure at least two-thirds of your devices are powered on.
- Call `ibpad` (or `ibsad`, if necessary) to match the configured address to the device switch settings.
- Reduce the write byte count to that which is expected by the Controller.

# EADR (3)

EADR occurs when the GPIB board is CIC and is not properly addressing itself before read and write functions. This error is usually associated with board-level functions.

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.

## Solutions

- Make sure that the GPIB board is addressed correctly before calling `ibrd`, `ibwrt`, `RcvRespMsg`, or `SendDataBytes`.
- 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
- `ibpad` or `ibsad` called with invalid addresses
- `ibppc` called with invalid parallel poll configurations
- A board-level NI-488 call made with a valid device descriptor or a device-level NI-488 call made with a board descriptor
- An NI-488.2 routine called with an invalid address
- `PPollConfig` called with an invalid data line or sense bit

## Solutions

- Make sure that the parameters passed to the NI-488 function or NI-488.2 routine are valid.
- Do not use a device descriptor in a board function or vice-versa.

# ESAC (5)

ESAC results when `ibsic`, `ibsre`, `SendIFC`, or `EnableRemote` is called when the GPIB board does not have System Controller capability.

## Solutions

Give the GPIB board System Controller capability by calling `ibrsc 1` or by using `ibconf` to configure that capability into the software.

# EABO (6)

EABO indicates that an I/O operation has been canceled, usually due to a timeout condition. Other causes for this error are calling `ibstop` or receiving the Device Clear message from the CIC while performing an I/O operation.

Frequently, the I/O is not progressing (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.

### Solutions

- Use the correct byte count in input functions or have the Talker use the END message to signify the end of the transfer.
- Lengthen the timeout period for the I/O operation using `ibtmo`.
- Make sure that you have configured your device to send data before you request data.

## ENEB (7)

ENEB occurs when no GPIB board exists at the I/O address specified in the configuration program. This problem 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, there is a system conflict with the base I/O address, or the `Use This GPIB Interface` field is set incorrectly in `ibconf`.

### Solutions

- Make sure there is a GPIB board in your computer that is properly configured both in hardware and software using a valid base I/O address.
- Make sure the `Use This GPIB Interface` field in `ibconf` is set to `yes`.

## EOIP (10)

EOIP occurs when an asynchronous I/O operation has not finished before some other call is made. During asynchronous I/O, you can only use `ibstop`, `ibwait`, and `ibonl`, or perform other non-GPIB operations. Once the asynchronous I/O has begun, further GPIB calls other than `ibstop`, `ibwait`, or `ibonl` are strictly limited. If a call might interfere with the I/O operation in progress, the driver returns EOIP.

### Solutions

Resynchronize the driver and the application before making any further GPIB calls. Resynchronization is accomplished by using one of the following three functions:

- `ibwait` If the returned `ibsta` contains CMPL then the driver and application are resynchronized.
- `ibstop` The I/O is canceled; the driver and application are resynchronized.
- `ibonl` The I/O is canceled and the interface is reset; the driver and application are resynchronized.

# ECAP (11)

ECAP results when your GPIB board lacks the ability to carry out an operation or when a particular capability has been disabled in the software and a call is made that requires the capability.

## Solutions

Check the validity of the call, or make sure your GPIB interface board and the driver both have the needed 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 `ibcntl`.

## Solutions

- Make sure the filename, path, and drive that you specified are correct.
- Make sure that the access mode of the file is correct.
- Make sure there is enough room on the disk to hold the file.

# 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 default configuration or the `ibtmo` function. EBUS results if a timeout occurred while sending these command bytes.

## Solutions

- Verify that the instrument is operating correctly.
- Check for loose or faulty cabling or several powered-off instruments on the GPIB.
- If the timeout period is too short for the driver to send command bytes, increase the timeout period.

# ESTB (15)

ESTB is reported only by the `ibrsp` function. ESTB indicates that one or more serial poll status bytes received from automatic serial polls have been discarded because of a lack of storage space. Several older status bytes are available; however, the oldest is being returned by the `ibrsp` call.

## Solutions

- Call `ibrsp` more frequently to empty the queue.
- Disable autopolling with the `ibconfig` function or the `ibconf` utility.

# ESRQ (16)

ESRQ occurs only during the `ibwait` function or the `WaitSRQ` routine. ESRQ indicates that a wait for RQS is not possible because the GPIB SRQ line is stuck on. This situation can be caused by the following events:

- Usually, a device unknown to the software is asserting SRQ. Because the software does not know of this device, it can never serial poll the device and unassert SRQ.
- A GPIB bus tester or similar equipment might be forcing the SRQ line to be asserted.
- A cable problem might exist involving the SRQ line.

Although the occurrence of ESRQ warns you of a definite GPIB problem, it does not affect GPIB operations, except that you cannot depend on the RQS bit while the condition lasts.

## Solutions

Check to see if other devices not used by your application are asserting SRQ. Disconnect them from the GPIB if necessary.

# ETAB (20)

ETAB occurs only during the `FindLstn`, `FindRQS`, and `ibevent` functions. ETAB indicates that there was some problem with a table used by these functions.

- In the case of `FindLstn`, ETAB means that the given table did not have enough room to hold all the addresses of the Listeners found.
- In the case of `FindRQS`, ETAB means that none of the devices in the given table were requesting service.
- In the case of `ibevent`, ETAB means the event queue overflowed and event information was lost.

## Solutions

In the case of `FindLstn`, increase the size of result arrays. In the case of `FindRQS`, check to see if other devices not used by your application are asserting SRQ. Disconnect them from the GPIB if necessary. In the case of ETAB returned from `ibevent`, call `ibevent` more often to empty the queue.

# Appendix C
# Universal Language Interface

This appendix describes how to install and use the Universal Language Interface (ULI).

## Overview

If you have an existing program that uses HP-style calls, you can use the Universal Language Interface (ULI) to access the NI-488.2 driver. DOS file calls directly access the ULI through the standard I/O commands of a programming language, eliminating the need for a language interface. However, the ULI is a low-performance interface to the NI-488.2 driver. If possible, use the NI-488.2 language interfaces for high performance and maximum flexibility.

The ULI driver file `uli.com` is included with your GPIB interface board. The ULI driver is a terminate-and-stay-resident (TSR) utility that interfaces between DOS and the NI-488.2 driver `gpib.com`.

## Installing the ULI

Before you install the ULI software, you must already have your GPIB board and NI-488.2 software installed and configured.

To install the ULI software, follow these steps:

1. Change to the ULI subdirectory that was created when you installed the NI-488.2 software.

2. Enter the `uli` command as follows:

   ```
   uli
   ```

A message displays on the computer screen that explains the results of loading the ULI driver into memory. The ULI driver stays resident in memory until the computer is turned off or restarted.

**Note:** ***While the ULI driver is loaded, you cannot use the standard NI-488 functions and NI-488.2 routines.***

# ULI Example Sequence

The following paragraphs demonstrate how to use the ULI with a generic instrument in a BASIC program. This instrument has a primary address of 2 and no secondary address. The techniques used in this program are general techniques that apply to the control of most instruments.

## Initializing the System

1. To establish communication with the ULI driver from a programming language such as BASIC, use the `OPEN` statement. In BASIC, two `OPEN` statements are required: one for output and the other for input. The ULI driver uses the board names defined in ibconf for the NI-488.2 driver. Use the following two commands in your BASIC program:

   ```
   100     OPEN "GPIB0" FOR OUTPUT AS #1
   110     OPEN "GPIB0" FOR INPUT AS #2
   ```

2. The next step is to initialize the bus by sending the Interface Clear (IFC) message. Enter the following command:

   ```
   120     PRINT #1, "ABORT"
   ```

3. To examine the status of the bus, use the status command as follows:

   ```
   130  PRINT #1, "STATUS"
   140  INPUT #2, IBSTA%,IBERR%,IBCNT%
   150  PRINT IBSTA%,IBERR%,IBCNT%
   ```

   `IBSTA%` contains the status of the last bus call, while `IBERR%` contains an error message if an error condition exists. `IBCNT%` contains the number of bytes of data successfully transmitted across the bus. These variables are described more fully in Chapter 3, *Developing Your Application*.

## Configuring the Device

After you have initialized the bus, you can initialize the device.

1. To place the instrument in remote mode, enter the `REMOTE` command:

   ```
   160     PRINT #1, "REMOTE 2"
   ```

   The `2` after `REMOTE` is the address of the device to be placed in remote mode.

2. Now that the instrument is ready to receive commands, the OUTPUT command sends any device-specific commands. For example, sending F1R0S2 to the instrument is done as follows:

```
170     PRINT #1, "OUTPUT 2#6; F1R0S2"
```

   The OUTPUT command sends the six bytes of data, F1R0S2, to the specified device (in this case, device with address 2). The address can be just a primary address, such as 2 or 5, or it can include a secondary address, such as 0201.

### Taking Readings

Once the instrument is in the operating mode, it can take a reading and display it as follows:

```
180         PRINT #1,    "ENTER 2"
190         INPUT #2,    RD$
200         PRINT RD$
```

The ENTER command takes an address with an optional secondary address and programs the driver to read from the instrument. The data is stored in the BASIC variable RD$ when the INPUT statement is executed. With BASIC, a maximum of 255 bytes of data can be read into a string variable. If the device returns more than 255 bytes, use multiple INPUT statements to bring the data into BASIC.

After you take readings from the instrument, you can use any BASIC string operations to manipulate the data.

### Handling Errors

BASIC has a method for handling errors encountered during the operation of the IEEE 488 bus with its ON ERROR GOTO command. If you prefer to write your own error-handling routine with the ON ERROR command, turn off the ULI automatic error detection and make sure that the applications monitor is not installed. You can turn off the ULI automatic error detection by entering the following command:

```
10          PRINT #1, "ERRTRAP OFF"
```

Include a service routine and the code for handling the error when it occurs. The following code segment can be used to implement ON ERROR. An error value of 24 indicates a GPIB error. All other error values reference non-GPIB errors.

```
50          ON ERROR GOTO 4000

4000        'SERVICE ROUTINE FOR ERRORS
4010        IF ERR <> 24 THEN GOTO 4090
4020        ELSE CLOSE
```

```
4030       OPEN "GPIB0" FOR INPUT AS #1
4040       OPEN "GPIB0" FOR OUTPUT AS #2
4050       PRINT #1, "STATUS"
4060       INPUT #2, IBSTA%,IBERR%,IBCNT%
4070       PRINT IBSTA%,IBERR%,IBCNT%
4080       RESUME NEXT
4090       PRINT "NON-GPIB ERROR"
5000       RESUME NEXT
```

# Data Transfer Termination

GPIB data transfers are two-part operations. Each part must include a terminator, which indicates the end of the transmission. One part occurs between your application program and the NI-488.2 driver, and includes a language terminator. The other part of the transfer occurs between the driver and the GPIB device, and includes a GPIB terminator. With the ULI, you have the flexibility to configure your transfer terminators to match the requirements of any combination of programming languages and devices. The following paragraphs discuss each type of terminator and the implications for language applications.

# OUTPUT Terminators

## Language Terminator

When receiving I/O commands from the application program, the ULI uses the language terminator to recognize the end of the string. In BASIC you can terminate all strings and numbers either by sending the language terminator with an I/O call or by appending the language terminator to the output string.

The language terminator may be one or two characters in length. The default language terminator, `CR LF`, works for most languages. If, however, you need to change it, use the `langeos` function. If you change the language terminator to a value that does not work with your programming language, your application either does not read the data correctly or does not terminate the data transfer.

## GPIB Terminator

When the ULI passes the I/O commands to the device, the language terminator is replaced by the GPIB terminator. If you want to send the same characters to the device that are used as a language terminator, set the GPIB terminator to the same value as the language terminator. For example, to send `CR LF` to the device, use the following function call:

```
PRINT #1, "GPIBEOS OUT CR LF"
```

The default setting of the GPIB terminator is disabled–that is, no termination characters are sent.

Figure C-1 shows an example of an OUTPUT command that replaces the language terminator with a specified GPIB terminator.

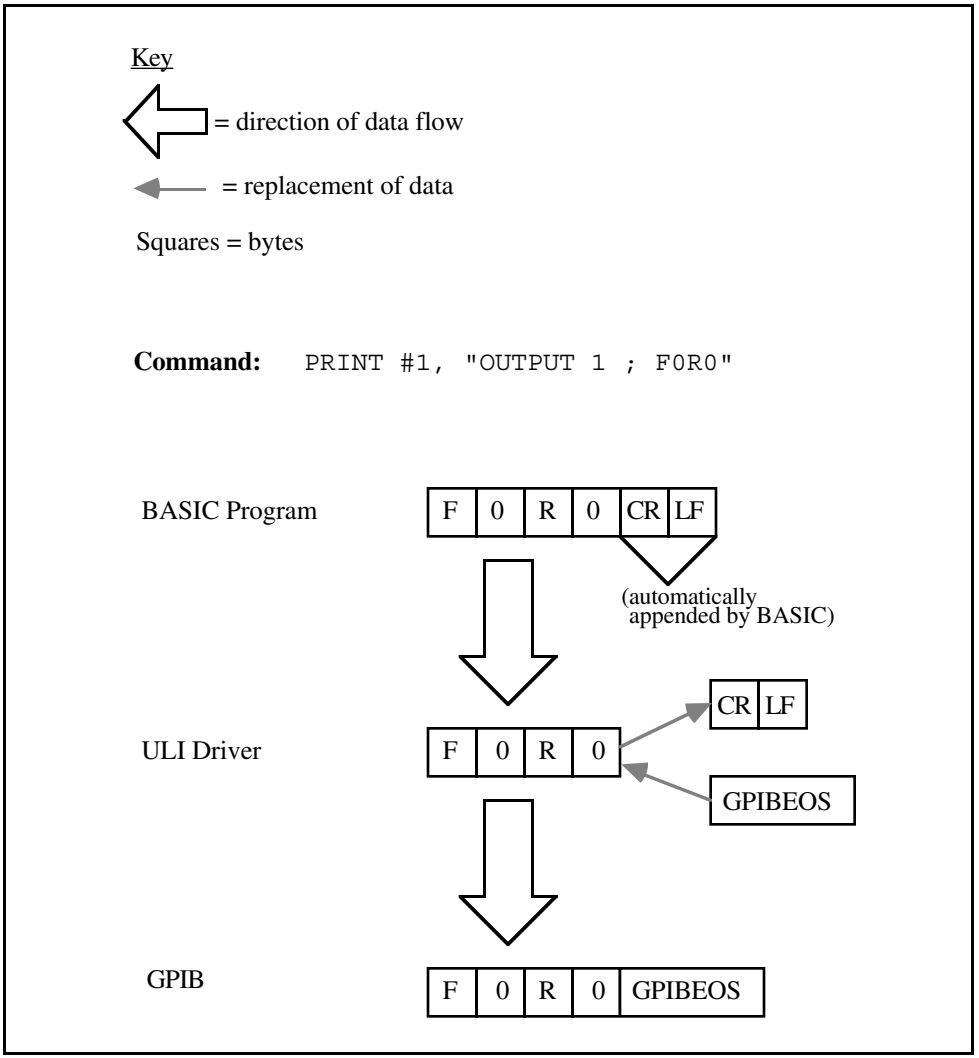



Figure C-1. Character Count Not Included in the OUTPUT Statement

Instead of using a GPIB terminator, you can specify a count value in an output statement. specifying a byte count automatically disables the GPIB terminator. The following BASIC statement sends the characters `F`, `0`, `R`, and `2` to the GPIB device.

```
PRINT #1,"OUTPUT 1#4;F0R2"
```

This command does *not* send a GPIB terminator because the count value of four does not leave enough space for a terminator character. If the count had been six or more (five for a one-character GPIB terminator), a GPIB terminator would have been included.

# INPUT Terminators

## Language Terminator

The INPUT language terminator operates similar to the OUTPUT language terminator, except that the direction of the data flow is reversed. The GPIB terminator is replaced by the language terminator, which is appended to the data before being sent to the application program.

You must sometimes treat data as binary information, rather than as formatted ASCII strings or numeric values. For example, if you specify a count value in your application, you receive only as many characters as you requested, and you might not get the language terminator required by your programming language. In this case, use the binary transfer functions of the programming language (such as `INPUT$` in BASIC). Refer to the next section, *GPIB Terminator*, for examples.

## GPIB Terminator

Because each instrument manufacturer can use a different method for indicating the completion of a data string, you might need to use a different GPIB terminator for each type of device connected to your computer.

Device input terminates under the following four conditions:

- A predefined count is reached
- A one-character GPIB terminator is defined and received
- A two-character GPIB terminator is defined and received in the correct order
- The GPIB EOI line is asserted with the last data byte

The following paragraphs describe each of the conditions.

**Note:** ***In each condition, the letters on the data line represent bytes of data that you can enter into the*** `INPUT` ***string.***

Condition 1. A predefined count is reached.

Figure C-2 shows how condition 1 is handled by the ULI driver.

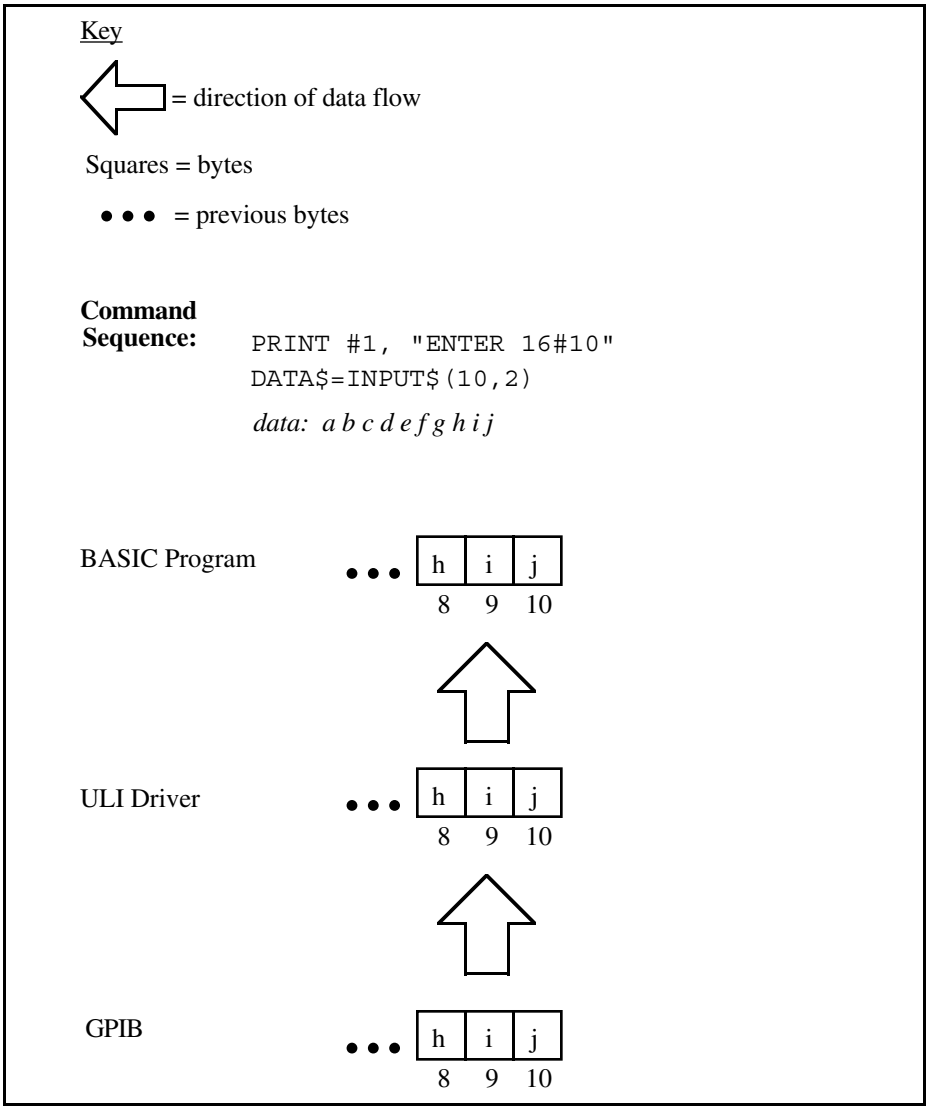



Figure C-2. Character Count Included in the ENTER Statement

Condition 2. A one-character GPIB terminator is defined and received.

Example:

```
PRINT #1,"GPIBEOS IN CR"
PRINT #1,"ENTER 16"
INPUT #2,DATA$
```

*data: a b c d e CR*

Condition 3. A two-character GPIB terminator is defined and received in the correct order.

Example:

```
PRINT #1,"GPIBEOS IN CHR(\x30)
            CHR(\x31)"
PRINT #1,"ENTER 16"
INPUT #2,DATA$
```

*data: a b c d e f 0 1 (ASCII 0 = hex 30, ASCII 1 = hex 31)*

**Note:** ***If either hex 30 or hex 31 is received, but not both (or if hex 31 and hex 30 are received in that order), the input does not terminate. If hex 30 and hex 31 are received in that order, the input terminates*** .

Figure C-3 shows how conditions 2 and 3 are handled by the ULI driver.

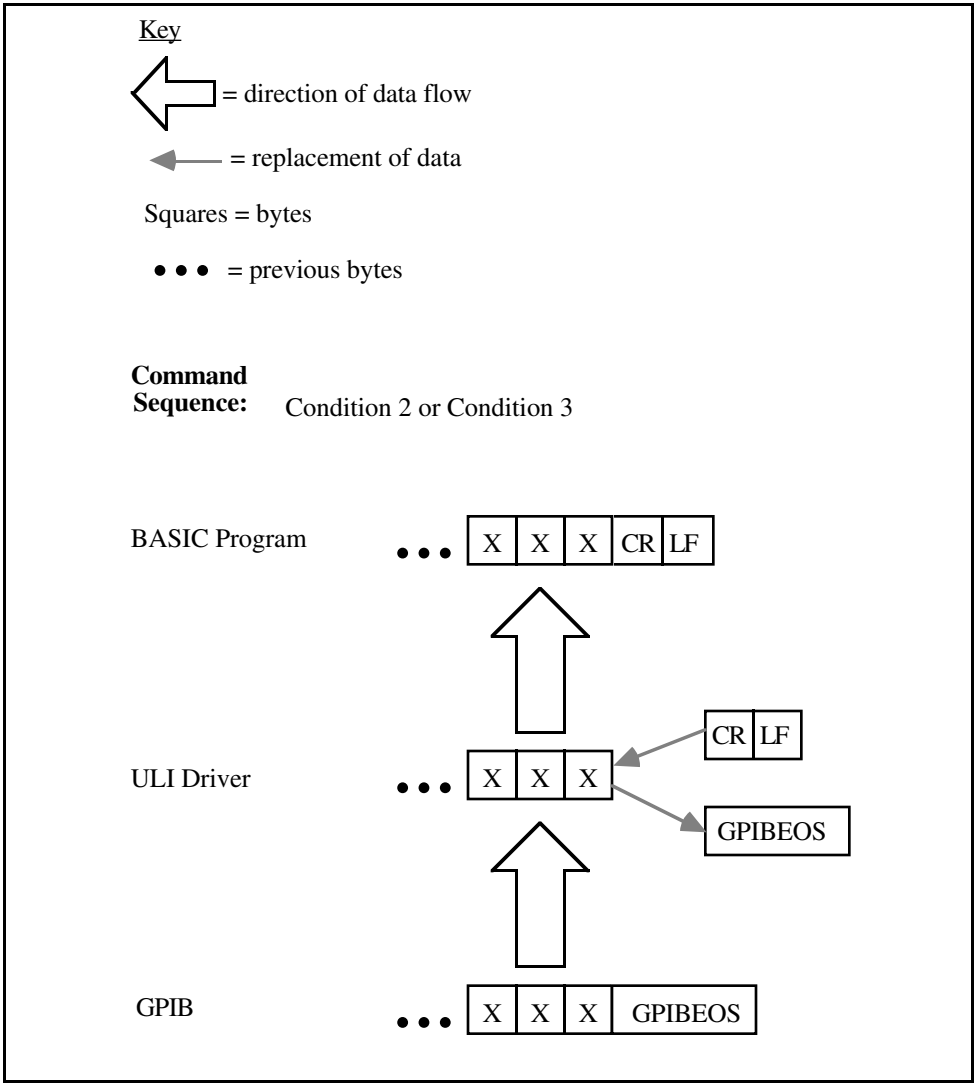



Figure C-3. Character Count Not Included in the ENTER Statement

Condition 4. The GPIB EOI line is asserted with the last data byte.

Example:

```
PRINT #1,ENTER 16"
INPUT #2,DATA$
```

*data: a b c d e f g h*

EOI is asserted with the last character (*h*).

# ULI Functions

Table C-1 contains a complete list of the ULI functions and a short description of each.

Table C-1. Universal Language Interface Functions

| Function | Description |
|---|---|
| ABORT | Initializes the bus and terminates a bus transfer if a transfer is in progress |
| CLEAR | Clears selected devices |
| ENTER | Reads data from the GPIB |
| ERRTRAP | Enables or disables automatic error trapping |
| GPIBEOS | Selects the termination characters for GPIB data |
| LANGEOS | Sets the termination characters for data passed to and from an application program |
| LOCAL | Places the addressed devices or all devices in local mode |
| LOCAL LOCKOUT | Inhibits all devices from being controlled from their front panel |
| OFFLINE | Places the device or interface board offline |
| ONLINE | Places the interface board online |
| OUTPUT | Sends data to a device or multiple devices |
| PASS CONTROL | Passes control to another controller on bus |
| PPOLL | Conducts a parallel poll of devices |
| PPOLL CONFIGURE | Configures parallel poll response of devices |

(continues)

Table C-1. Universal Language Interface Functions (Continued)

| Function | Description |
|---|---|
| PPOLL RESPONSE | Sets the ist bit on the GPIB interface board for parallel poll responses |
| PPOLL UNCONFIGURE | Disables the parallel poll response of devices |
| REMOTE | Sets the REN bus line and places devices in remote mode |
| REQUEST | Requests service from the Controller |
| RESET | Sets the configuration parameters to default values |
| SEND | Sends GPIB management commands |
| SPOLL | Serial polls a device |
| STATUS | Returns the status of the previous bus call |
| TIMEOUT | Sets the time limit |
| TRIGGER | Triggers selected device/devices |

## Function Syntax Conventions

The following conventions are used in the syntax descriptions for the ULI functions:

| | |
|---|---|
| 255 Characters | No command or *data* part of the command can be more than 255 characters long. |
| UPPERCASE TEXT | Items in uppercase text, such as ENTER or ABORT, must be entered exactly as shown. |
| Blank space | Blank spaces in commands are generally ignored. For example, LOCAL LOCKOUT is the same as LOCALLOCKOUT. Spaces are *not* ignored in two places: the data part of an OUTPUT command or between the arguments of a SEND command. |
| # and ; | Enter the # character exactly as shown. Enter the ; character exactly as shown. |
| [ ] | Items enclosed in square brackets are optional. |
| \| | Multiple items enclosed in square brackets separated by vertical lines ([item1] \| [item2] \| [item3]) are optional. You can choose one option or no option. |

< > Items enclosed in angle brackets must be replaced by an appropriate value.

## Function Descriptions

The remainder of this chapter contains a detailed description of each ULI function with at least one example. The functions are listed in alphabetical order for easy reference.

### ABORT

**Purpose**: Initializes the bus and terminates a bus data transfer if a transfer is in progress.

**Format**: `ABORT`

**Response**: None

**Remarks**: The `ABORT` command causes the GPIB board to become System Controller (SC) and Controller-In-Charge (CIC). It then asserts the Interface Clear (IFC) line followed by the Attention (ATN) line, which stops any bus transaction.

Using the `ABORT` command to initialize the bus is a shortcut to the normal procedure.

**Example**: Initialize the bus or terminate a bus data transfer.

```
PRINT #1, "ABORT"
```

### CLEAR

**Purpose**: Clears selected devices.

**Format**: `CLEAR [<addr>[,<addr>]]`

`addr` is the primary address of a device (and optional secondary address). A comma ( , ) separates multiple addresses.

**Response**: None

**Remarks**: The `CLEAR` command clears the internal or device functions of a specified device by sending the Selected Device Clear (SDC) message or Device Clear (DCL) if no address is present.

**Examples**: 1. Clear all devices.

```
PRINT #1, "CLEAR"
```

2. Clear device 2.

```
PRINT #1, "CLEAR 2"
```

---

## ENTER

**Purpose**: Reads data from the GPIB.

**Format**: `ENTER [<addr>] [# <count>] [DMA] Preserve Blanks`

`addr` is the IEEE-bus device address. `count` is the number of characters to read. `DMA` turns Direct Memory Access (`DMA`) on. `Preserve Blanks` preserves leading blanks of a string variable that may be part of a device's output format. BASIC normally removes these blanks.

**Response**: Device-specific data

**Remarks**: If a device address is present, that device is addressed to talk. If no address is present, the GPIB board must already be configured to receive data, either as a result of an immediately preceding `ENTER` command or as a result of a `SEND` command. If the count is specified, `count` characters are read from the device. Otherwise, the subsequent `INPUT` command terminates upon detection of the `GPIBEOS` input terminator.

**Examples**: 1. Read a line of data from device 2.

```
PRINT #1, "ENTER 2"
LINE INPUT #2, A$
```

The `Line Input` statement ignores delimiters in the input stream. See the BASIC reference manual that is supplied with your version of BASIC for additional information.

2. Read 10 bytes of data from device 16:

```
PRINT #1, "ENTER 16#10"
B$ = INPUT$(10,2)
```

3. Read data using `GPIBEOS` input terminator:

```
PRINT #1, "ENTER 2"
INPUT #2, A$
```

## ERRTRAP

**Purpose**: Enables or disables automatic error trapping.

**Format**: `ERRTRAP [ON|OFF]`

**Response**: None

**Remarks**: If `ERRTRAP` is disabled, the only error message that appears on your screen is `Device I/O Error.` You must enable `ERRTRAP` to receive GPIB-specific error messages (such as No Listener). The default setting for this feature is ON (enabled).

**Example**: Disable automatic error detection.

```
PRINT #1, "ERRTRAP OFF"
```

## GPIBEOS

**Purpose**: Selects the termination characters for GPIB data.

**Format**: `GPIBEOS [IN|OUT] <term>`

`IN` means input terminator. `OUT` means output terminator. If neither `IN` nor `OUT` are present, then both terminators are set. `term` is the character that will be used as a terminator.

**Response**: None

**Remarks**: Refer to the section *Data Transfer Termination* in this appendix. `term` can be any of the following:

| | |
|---|---|
| `CR` | Carriage Return |
| `LF` | Line Feed |
| `CHR(#)` | # is integer between 0 and 255 |
| `'X` | X is any printable ASCII character |

The `GPIBEOS` terminator assignments are canceled if this command is executed without parameters.

**Examples**: 1. Set both terminators to CR.

```
PRINT #1, "GPIBEOS CR"
```

2. Set the input terminator to LF.

```
PRINT #1, "GPIBEOS IN LF"
```

3. Set the output terminator to CR LF.

```
PRINT #1, "GPIBEOS OUT CR LF"
```

## LANGEOS

**Purpose**: Sets the termination characters for data passed to and from an application program.

**Format**: `LANGEOS [IN|OUT] <term>`

`IN` means input terminator. `OUT` means output terminator. If neither `IN` nor `OUT` are present, both terminators are set the same. `term` is the character that will be used as a terminator.

**Response**: None

**Remarks**: Refer to the section *Data Transfer Termination* in this appendix. `term` can be any of the following:

| | |
|---|---|
| `CR` | Carriage Return |
| `LF` | Line Feed |
| `CHR(#)` | # is integer between 0 and 255 |
| `'X` | X is any printable ASCII character |

**Examples**: 1. Set both terminators to CR.

```
PRINT #1, "LANGEOS CR"
```

2. Set the input terminator to LF.

```
PRINT #1, "LANGEOS IN LF"
```

3. Set the output terminator to CR LF.

```
PRINT #1, "LANGEOS OUT CR LF"
```

## LOCAL

**Purpose**: Places the addressed device/devices or all devices in local mode.

**Format**: `LOCAL [<addr>[,<addr>]]`

`addr` is the primary address of the device (optional secondary address). A comma ( , ) separates multiple primary addresses.

**Response**: None

**Remarks**: The Go To Local (GTL) message is sent to the device, placing it in manual operation mode if an address is present; otherwise, the Remote Enable line is unasserted.

**Examples**: 1. Unassert the Remote Enable line.

```
PRINT #1, "LOCAL"
```

2. Send Go To Local to devices `2` and `5`.

```
PRINT #1, "LOCAL 2,5"
```

## LOCAL LOCKOUT

**Purpose**: Inhibits all devices from being controlled from their front panel.

**Format**: `LOCAL LOCKOUT`

**Response**: None

**Example**: Send Local Lockout bus command.

```
PRINT #1, "LOCAL LOCKOUT"
```

## OFFLINE

**Purpose**: Places the device or interface board offline which is equivalent to disconnecting its GPIB cable from the other devices.

**Format**: `OFFLINE`

**Response**: None

**Example**: Place the device offline.

```
PRINT #1, "OFFLINE"
```

## ONLINE

**Purpose**: Places interface board online which restores the default configuration strings of a device or interface board.

**Format**: `ONLINE`

**Response**: None

**Example**: Place interface board online.

```
PRINT #1, "ONLINE"
```

## OUTPUT

**Purpose**: Sends data to a device or multiple devices.

**Format**: `OUTPUT [<addr>[,<addr>]] [# <count>] [DMA] [END|NOEND]; <data>`

`addr` is the address of the device (with an optional secondary address). `count` is the number of characters to send. `DMA` turns Direct Memory Access (`DMA`) on. `END|NOEND` determines whether or not the EOI line is asserted at the end of a data transmission. `data` is the information that is being sent to the device or devices. A semicolon ( `;` ) separates the command from the device-dependent data.

**Response**: None

**Remarks**: If a device address is present, that device is addressed to listen. If no address is present, the GPIB board must already be configured to transmit data, either as a result of an immediately preceding `OUTPUT` command or as a result of a `SEND` command. If the count is specified, that exact number of characters are sent to the device. Otherwise, the `OUTPUT` command adds the `GPIBEOS` output terminator.

**Examples**: 1. Send `CURV? <GPIBEOS>` to device `2`.

**Note:** ***You must have already sent the*** `GPIBEOS` ***command to set up the EOS termination character.***

```
PRINT #1, "OUTPUT 2;CURV?"
```

2. Send F0R0 to device 10, 11, and 12.

```
PRINT #1, "OUTPUT 10,11,12#4;F0R0"
```

## PASS CONTROL

**Purpose**: Passes control to another Controller on the bus.

**Format**: PASS CONTROL <addr>

addr is the address of a device.

**Response**: None

**Remarks**: The GPIB board can regain control by issuing an ABORT command.

**Example**: Control is passed to device 2.

```
PRINT #1, "PASS CONTROL 2"
```

## PPOLL

**Purpose**: Conducts a parallel poll of devices.

**Format**: PPOLL

**Response**: Number between 0 and 255 that is the parallel poll response.

**Remarks**: Devices on the GPIB bus are polled in parallel (up to eight devices). Not every GPIB device supports parallel polls.

**Example**: Conduct a parallel poll.

```
PRINT #1 "PPOLL"

INPUT #2, PPRESP%
```

## PPOLL CONFIGURE

**Purpose**: Configures the parallel poll response of devices.

**Format**: `PPOLL CONFIGURE <addr> ; <number>`

`addr` is the primary address of the device to be configured. A semicolon ( `;` ) separates the command from the device-dependent data. `number` is the pattern that specifies which line is asserted by the device on a parallel poll and whether the parallel poll will be enabled or disabled.

**Response**: None

**Remarks**: Refer to Chapter 7, *GPIB Programming Techniques*, for more information on parallel polling.

**Example**: Send decimal `41` to device `2` for configuring the parallel poll.

```
PRINT #1, "PPOLL CONFIGURE 2;41"
```

## PPOLL RESPONSE

**Purpose**: Sets the individual status ( `ist` ) bit on the GPIB interface board for parallel poll responses.

**Format**: `PPOLL RESPONSE <number>`

`number` indicates status of the GPIB interface board. If `number` is non-zero, the `ist` bit is set. If `number` is zero, the `ist` bit is cleared.

**Response**: None

**Remarks**: Refer to Chapter 7, *GPIB Programming Techniques*, for more information on parallel polling.

**Example**: Sets the individual status bit.

```
PRINT #1, "PPOLL RESPONSE 2"
```

## PPOLL UNCONFIGURE

**Purpose**: Disables the parallel poll response for devices.

**Format**: `PPOLL UNCONFIGURE [<addr>[,<addr>]]`

**Response**: None

**Examples**: 1. Unconfigure device `2`.

```
PRINT #1, "PPOLL UNCONFIGURE 2"
```

2. Unconfigure all devices.

```
PRINT #1, "PPOLL UNCONFIGURE"
```

## REMOTE

**Purpose**: Sets the Remote Enable (REN) bus line and places devices in remote mode.

**Format**: `REMOTE [<addr>[,<addr>]]`

`addr` is the primary address of a device (optional secondary address). A comma ( `,` ) separates multiple primary addresses.

**Response**: None

**Remarks**: The REN bus line is asserted. If a device address is specified, that device is also addressed to listen in order to place it in remote programming mode.

**Examples**: 1. Assert Remote Enable.

```
PRINT #1, "REMOTE"
```

2. Assert Remote Enable and address devices `2` and `5` to listen.

```
PRINT #1, "REMOTE 2, 5"
```

## REQUEST

**Purpose**: Request service and/or set or change the serial poll status byte.

**Format**: `REQUEST <number>`

`number` 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) of the number is set, the GPIB board additionally requests service from the Controller by asserting the GPIB SRQ line.

**Response**: None

**Remarks**: The `REQUEST` command 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.

**Example**: Request service with a status byte of `&H41`.

```
PRINT #1, "REQUEST &H41";
```

## RESET

**Purpose**: Sets the configuration parameters (primary and secondary address, `LANGEOS`, and `GPIBEOS` characters) to their default values.

**Format**: `RESET`

**Response**: None

**Example**: Restore address and EOS parameters.

```
PRINT #1, "RESET"
```

## SEND

**Purpose**: Sends GPIB management commands.

**Format**:

```
SEND [MTA|TALK <addr>] [MLA|LISTEN <addr>] [UNL]
[UNT] [DATA <number>
[, <number>, <...>]]
```

MTA is the talk address of the GPIB board. TALK addr combines the talk address of a device with primary address addr. MLA is the listen address of the GPIB board. LISTEN addr combines the listen address of a device with primary address addr. UNL is the unlisten command message. It is always defined to be &H3F. UNT is the untalk command message. It is always defined to be &H5F. DATA is the device-specific data to be transferred. number is an integer between 0 and 255. A comma ( , ) separates multiple primary addresses.

**Response**: None

**Remarks**: Gives byte-by-byte control of data and transfers on the bus. These commands give the maximum flexibility and control to the GPIB controller.

**Example**: Send two data bytes, a decimal 1 and a decimal 2, to the device at address 1 from the device at address 2.

```
PRINT #1, "SEND LISTEN 1 TALK 2 DATA 1,2"
```

## SPOLL

**Purpose**: Serial polls a device.

**Format**: SPOLL [<addr>]

addr is the primary address of a device.

**Response**: An integer between 0 and 255 that is the Serial Poll response of the device.

**Remarks**: Refer to Chapter 7, *GPIB Programming Techniques*, for more information on serial polling.

**Examples**:

1. Serial poll device 2.

```
PRINT #1, "SPOLL 2"
```

2. Receive the serial poll status.

```
INPUT #2, SP%
```

## STATUS

**Purpose**: Returns the status of the previous bus call.

**Format**: `STATUS`

**Response**: Returns the status variables: `IBSTA%`, `IBERR%`, and `IBCNT%`.

**Remarks**: `IBSTA%`, `IBERR%`, and `IBCNT%` correspond to status, error, and count variables, respectively, as described in Chapter 3, *Developing Your Application*.

**Example**: Check status

```
PRINT #1,"STATUS"

INPUT #2, IBSTA%, IBERR%, IBCNT%.
```

## TIMEOUT

**Purpose**: Sets the time limit.

**Format**: `TIMEOUT <number>`

`number` is the code for the length of time for a timeout condition to occur.

**Response**: None

**Remarks**: Each data or control transaction on the bus must be completed within a certain time limit. This limit can be set to any one of the limits shown in Table C-2.

Table C-2. ULI Timeout Code Values

| Number | Minimum Timeout |
|---|---|
| 0 | disabled |
| 1 | 10 µs |
| 2 | 30 µs |
| 3 | 100 µs |
| 4 | 300 µs |
| 5 | 1 ms |
| 6 | 3 ms |
| 7 | 10 ms |
| 8 | 30 ms |
| 9 | 100 ms |
| 10 | 300 ms |
| 11 | 1 s |
| 12 | 3 s |
| 13 | 10 s |
| 14 | 30 s |
| 15 | 100 s |
| 16 | 300 s |
| 17 | 1000 s |

**Example**: Set the timeout period to 10 s.

```
PRINT #1, "TIMEOUT 13"
```

## TRIGGER

**Purpose**: Triggers selected device/devices.

**Format**: `TRIGGER [<addr>[,<addr>]]`

`addr` is the primary address of a device (optional secondary address). A comma ( , ) separates multiple addresses.

**Response**: None

**Remarks**: The `TRIGGER` command sends a Group Execute Trigger (GET) message to each specified device. If no devices are specified, GET is received only by those devices previously addressed to listen.

**Example**: Issue a `TRIGGER` command to devices `2` and `5`.

```
PRINT #1 "TRIGGER 2, 5"
```

# Appendix D
# Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *Technical Support Form* before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

**Corporate Headquarters**
(512) 795-8248
Technical support fax: (800) 328-2203
(512) 794-5678

| **Branch Offices** | **Phone Number** | **Fax Number** |
|---|---|---|
| Australia | 03 9 879 9422 | 03 9 879 9179 |
| Austria | 0662 45 79 90 0 | 0662 45 79 90 19 |
| Belgium | 02 757 00 20 | 02 757 03 11 |
| Canada (Ontario) | 519 622 9310 | 519 622 9311 |
| Canada (Quebec) | 514 694 8521 | 514 694 4399 |
| Denmark | 45 76 26 00 | 45 76 71 11 |
| Finland | 90 527 2321 | 90 502 2930 |
| France | 1 48 14 24 24 | 1 48 14 24 14 |
| Germany | 089 741 31 30 | 089 714 60 35 |
| Hong Kong | 2645 3186 | 2686 8505 |
| Italy | 02 48301892 | 02 48301915 |
| Japan | 03 5472 2970 | 03 5472 2977 |
| Korea | 02 596 7456 | 02 596 7455 |
| Mexico | 95 800 010 0793 | 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

Technical support is available at any time by fax. Include the information from the configuration form in your getting started manual. Use additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax ( ___ )_____ Phone ( ___ )_____

Computer brand _____

Model _____ Processor _____

Operating system _____

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 _____

(continues)

National Instruments software product _____

Version _____

Configuration _____

The problem is _____

_____

_____

_____

_____

_____

_____

_____

List any error messages _____

_____

_____

_____

_____

_____

_____

_____

The following steps will reproduce the problem _____

_____

_____

_____

_____

_____

_____

# 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™ User Manual for DOS**

Edition Date: **Jaunary 1996**

Part Number: **370904A-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
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678

# Glossary

| Prefix | Meaning | Value |
|---|---|---|
| n- | nano- | $10^{-9}$ |
| µ- | micro- | $10^{-6}$ |
| m- | milli- | $10^{-3}$ |
| k- | kilo- | $10^{3}$ |
| M- | mega- | $10^{6}$ |

## A

**acceptor handshake** — Listeners use this GPIB interface 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.

**ANSI** — American National Standards Institute.

**ASCII** — American Standard Code for Information Interchange.

**asynchronous** — An action or event that occurs at an unpredictable time with respect to the execution of a program.

**automatic serial polling (autopolling)** — A feature of the NI-488.2 software in which serial polls are executed automatically by the driver whenever a device asserts the GPIB SRQ line.

## B

**base I/O address** — See *I/O address*.

**BIOS** — Basic Input/Output System.

**board-level function** — A rudimentary function that performs a single operation.

# C

| | |
|---|---|
| CFE | Configuration Enable is the GPIB command which precedes CFGn and is used to place devices into their configuration mode. |
| CFGn | These GPIB commands (CFG1 through CFG15) follow CFE and are used to configure all devices for the number of meters of cable in the system so that HS488 transfers occur without errors. |
| CIC | See *Controller-In-Charge*. |
| Controller-In-Charge (CIC) | The device that manages the GPIB by sending interface messages to other devices. |
| CPU | Central processing unit. |

# D

| | |
|---|---|
| DAV (Data Valid) | One of the three GPIB handshake lines. See *handshake*. |
| DCL | Device Clear is the GPIB command used to reset the device or internal functions of all devices. See *SDC*. |
| Device Clear | See DCL. |
| device-level function | A 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. |
| DIO1 through DIO8 | The GPIB lines that are used to transmit command or data bytes from one device to another. |
| DMA (direct memory access) | High-speed data transfer between the GPIB board and memory that is not handled directly by the CPU. Not available on some systems. See *programmed I/O*. |
| driver | Device driver software installed within the operating system. |

## 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. |
| EOT | End of transmission. |
| ESB | The Event Status bit is part of the IEEE 488.2-defined status byte which is received from a device responding to a serial poll. |

## G

| | |
|---|---|
| GET | Group Execute Trigger is 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/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1987. |
| GPIB address | The address of a device on the GPIB, composed of a primary address (MLA and MTA) and an optional secondary address (MSA). The GPIB board has both a GPIB address and an I/O address. |
| GPIB board | Refers to the National Instruments family of GPIB interface boards. |
| Group Executed Trigger | See *GET*. |
| GTL | Go To Local is 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.<br><br>For more information about handshaking, refer to the ANSI/IEEE Standard 488.1-1987. |
| hex | Hexadecimal; a number represented in base 16, for example decimal 16 = hex 10. |
| high-level function | See *device-level function*. |
| Hz | Hertz. |

# I

| | |
|---|---|
| `ibcnt` | After each NI-488.2 I/O function, this global variable contains the actual number of bytes transmitted. |
| `ibconf` | The NI-488.2 driver configuration program for DOS. |
| `iberr` | A global variable that contains the specific error code associated with a function call that failed. |
| `ibic` | The Interface Bus Interactive Control program for DOS is used to communicate with GPIB devices, troubleshoot problems, and develop your application. |
| `ibsta` | At the end of each function call, this global variable (status word) contains status information. |
| IEEE | Institute of Electrical and Electronic Engineers. |
| interface message | A broadcast message sent from the Controller to all devices and used to manage the GPIB. |
| I/O (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. |

## K

| | |
|---|---|
| KB | Kilobytes. |

## L

| | |
|---|---|
| LAD (Listen Address) | See *MLA*. |
| language interface | Code that enables an application program that uses NI-488 functions or NI-488.2 routines to access the driver. |
| listen address | See *MLA*. |
| Listener | A GPIB device that receives data messages from a Talker. |
| low-level function | See *board-level function*. |

## M

| | |
|---|---|
| m | Meters. |
| MAV | The Message Available bit is part of the IEEE 488.2-defined status byte which is received from a device responding to a serial poll. |
| MB | Megabytes of memory. |
| memory-resident | Resident in RAM. |
| MLA (My Listen Address) | A GPIB command used to address a device to be a Listener. It can be any one of the 31 primary addresses. |

| | |
|---|---|
| MSA (My Secondary Address) | My Secondary Address is 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 secondary addresses for a total of 961 distinct listen or talk addresses for devices. |
| MTA (My Talk Address) | A GPIB command used to address a device to be a Talker. It can be any one of the 31 primary addresses. |
| multitasking | The concurrent processing of more than one program or task. |

## N

| | |
|---|---|
| NDAC (Not Data Accepted) | One of the three GPIB handshake lines. See *handshake*. |
| NRFD (Not Ready For Data) | One of the three GPIB handshake lines. See *handshake*. |

## P

| | |
|---|---|
| parallel poll | The process of polling all configured devices at once and reading a composite poll response. See *serial poll*. |
| PIO | See *programmed I/O*. |
| PPC (Parallel Poll Configure) | Parallel Poll Configure is the GPIB command used to configure an addressed Listener to participate in polls. |
| PPD (Parallel Poll Disable) | Parallel Poll Disable is the GPIB command used to disable a configured device from participating in polls. There are 16 PPD commands. |
| PPE (Parallel Poll Enable) | Parallel Poll Enable is 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 (Parallel Poll Unconfigure) | Parallel Poll Unconfigure is 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

| | |
|---|---|
| RAM | Random-access memory. |
| resynchronize | The NI-488.2 software and the user application must resynchronize after asynchronous I/O operations have completed. |
| RQS | Request Service. |

## S

| | |
|---|---|
| s | Seconds. |
| SDC | Selected Device Clear is 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*. |
| Service Request | See *SRQ*. |
| 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<br>(Serial Poll Disable) | Serial Poll Disable is the GPIB command used to cancel an SPE command. |
| SPE<br>(Serial Poll Enable) | Serial Poll Enable is the GPIB command used to enable a specific device to be polled. That device must also be addressed to talk. See *SPD*. |
| SRQ (Service Request) | The GPIB line that a device asserts to notify the CIC that the device needs servicing. |
| status byte | The IEEE 488.2-defined data byte sent by a device when it is serially polled. |
| status word | See *ibsta*. |
| synchronous | Refers to the relationship between the NI-488.2 driver functions and a process when executing driver functions is predictable; the process is blocked until the driver completes the function. |

| | |
|---|---|
| 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

| | |
|---|---|
| TAD (Talk Address) | See *MTA*. |
| Talker | A GPIB device that sends data messages to Listeners. |
| TCT | Take Control is 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` (unit descriptor) | 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. |
| ULI | Universal Language Interface |
| UNL | Unlisten is the GPIB command used to unaddress any active Listeners. |
| UNT | Untalk is the GPIB command used to unaddress an active Talker. |

# Index

## B



## C

## D

## E

# F



# G

## H



## I

## L

# M



# N

# O



# P



# Q

# R



# S

## T

# U

## V



## W