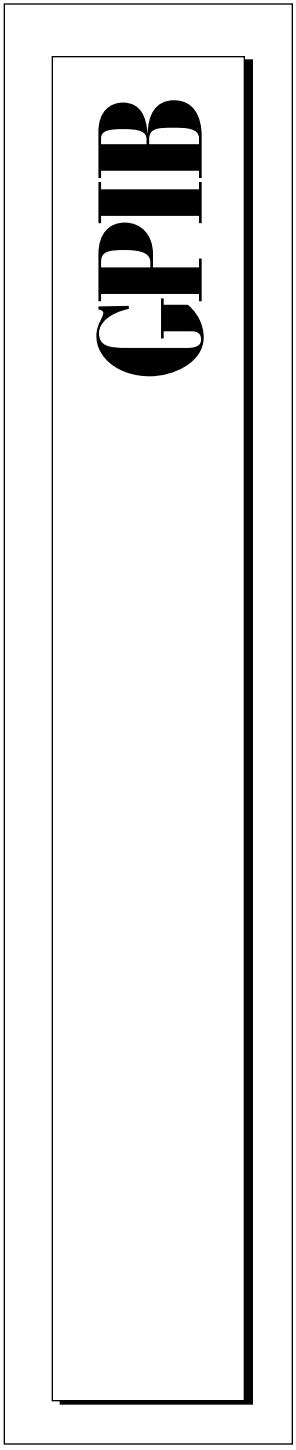


NI-488.2M™ User Manual for Windows 95

November 1995 Edition
Part Number 321037A-01

© Copyright 1995 National Instruments Corporation.
All Rights Reserved.

Internet Support

GPIB: gpib.support@natinst.com
DAQ: daq.support@natinst.com
VXI: vxi.support@natinst.com
LabVIEW: lv.support@natinst.com
LabWindows: lw.support@natinst.com
HiQ: hiq.support@natinst.com
VISA: visa.support@natinst.com

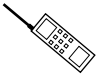
FTP Site: ftp.natinst.com
Web Address: www.natinst.com

Bulletin Board Support

BBS United States: (512) 794-5422 or (800) 327-3077
BBS United Kingdom: 01635 551422
BBS France: 1 48 65 15 59

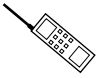
FaxBack Support

(512) 418-1111 or (800) 329-7177



Telephone Support (U.S.)

Tel: (512) 795-8248
Fax: (512) 794-5678 or (800) 328-2203



International 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 5 202 2544,
Netherlands 03480 33466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085,
Sweden 08 730 49 70, Switzerland 056 20 51 51, Taiwan 02 377 1200, U.K. 01635 523545

National Instruments Corporate Headquarters

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

Important Information

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

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

NI-488®, NI-488.2™, NI-488.2M™, and TNT4882C™ are trademarks of National Instruments Corporation.

Product and company names listed are trademarks or trade names of their respective companies.

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.

Table of Contents

About This Manual

Chapter 1 Introduction

Chapter 2
Application Examples

Chapter 3
Developing Your Application

Chapter 4
Debugging Your Application

Chapter 5
Win32 Interactive Control Utility

Chapter 6
GPIB Programming Techniques

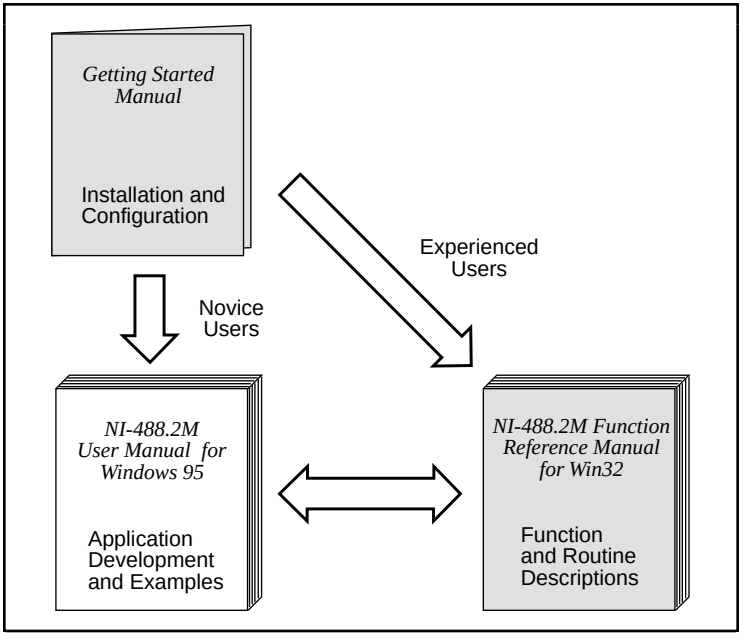
Figures

Tables

About This Manual

This manual describes the features and functions of the NI-488.2M software for Windows 95. The NI-488.2M software is meant to be used with the Microsoft Windows 95 operating system. This manual assumes that you are already familiar with Windows 95.

How to Use the Manual Set



Use the getting started manual to install and configure your GPIB hardware and software for Windows 95.

Use the *NI-488.2M User Manual for Windows 95* 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-488.2M Function Reference Manual for Win32* for specific NI-488 function and NI-488.2 routine information, such as format, parameters, and possible errors.

Organization of This Manual

This manual is organized as follows:

- Chapter 1, *Introduction*, gives an overview of GPIB and the NI-488.2M 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, *Win32 Interactive Control Utility*, introduces you to the interactive control program that you can use to communicate with GPIB devices interactively.
- Chapter 6, *GPIB Programming Techniques*, describes techniques for using some NI-488 functions and NI-488.2 routines in your application program.
- Chapter 7, *GPIB Configuration Utility*, contains a description of the software configuration program you can use to configure the NI-488.2M 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, *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 menus, menu items, or dialog box buttons or options.
italic	Italic text denotes emphasis, a cross reference, 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 are to be literally input from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, variables, filenames, and extensions, and for statements and comments taken from program code.
`bold monospace`	Bold lowercase text in this font denotes the messages and responses that the computer automatically prints to the screen.
`italic monospace`	Italic lowercase text in this font denotes that you must supply the appropriate words or values in the place of these items.
< >	Angle brackets enclose the name of a key on the keyboard–for example, <PageDown>.
-	A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys–for example, <Control-Alt-Delete>.
»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options»Substitute Fonts** directs you to pull down the **File** menu, select the **Page Setup** item, select **Options**, and finally select the **Substitute Fonts** option from the last dialog box.
IEEE 488 and IEEE 488.2	*IEEE 488* and *IEEE 488.2* refer to the ANSI/IEEE Standard 488.1-1987 and the ANSI/IEEE Standard 488.2-1987, respectively, which define the GPIB.
	Abbreviations, acronyms, metric prefixes, mnemonics, symbols, and terms are listed in the *Glossary*.

Related Documentation

The following document contains 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 Windows 95 User's Guide
- *Microsoft Win32 Software Development Kit for Microsoft Windows*

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 C, *Customer Communication*, at the end of this manual.

Chapter 1

Introduction

This chapter gives an overview of GPIB and the NI-488.2M 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, using a 3-wire handshake. 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 NI-488.2M 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 GPIB devices and boards must be assigned a unique GPIB address. A GPIB address is made up of two parts: a primary address and an optional secondary address.

The primary address is a number in the range 0 to 30. The GPIB Controller uses this 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 0-30)				

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 GPIB transfers.
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.

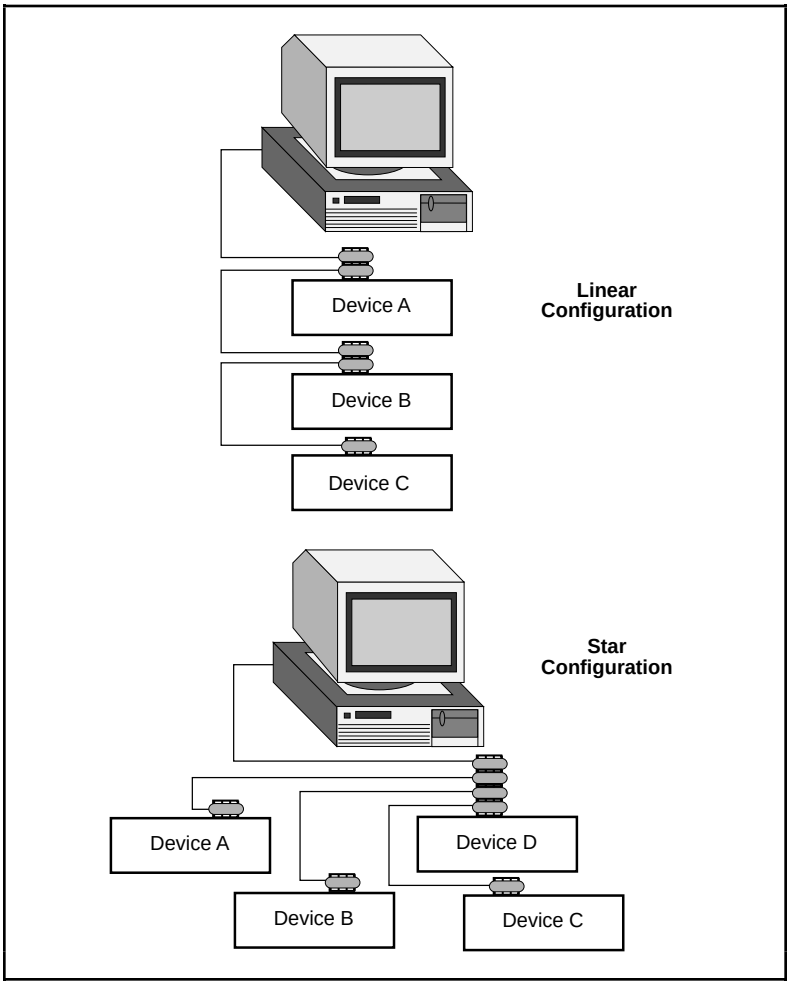


Figure 1-2. Linear and Star System Configuration

Controlling More Than One Board

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.

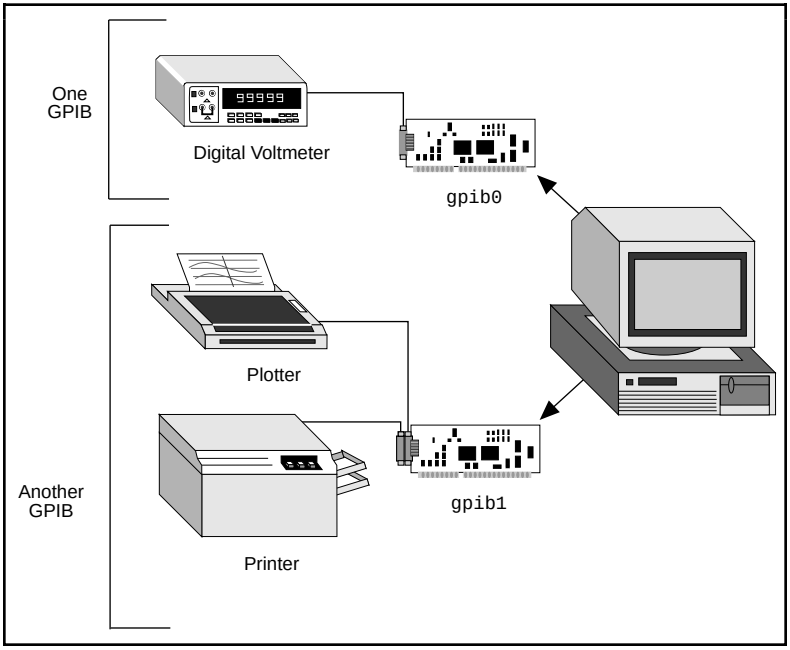


Figure 1-3. Example of Multiboard System Setup

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 as short as possible up to a maximum of 15 m of cable for each system.
- With 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 expander to increase the number of device loads. Extenders and expanders are available from National Instruments.

The following sections describe the NI-488.2M software, which controls the flow of communication on the GPIB.

The NI-488.2M Software Components

The following section highlights important components of the NI-488.2M software for Windows 95 and describes the function of each component.

NI-488.2M Driver and Driver Utilities

The distribution disk contains the following driver and utility files:

- `readme.txt` is a documentation file that contains important information about the NI-488.2M software and a description of any new features. Before you use the software, read this file for the most recent information.
- Native, 32-bit NI-488.2M driver components: These files are all of the form `gpib*.*`. They are a collection of dynamically loadable, Plug and Play aware, and multitasking aware virtual device drivers and dynamic link libraries. They are installed into the Windows System directory.
- `gpib-32.dll` is a Win32 dynamic link library that acts as the interface between all Windows 95 GPIB applications and the NI-488.2M driver components.
- Win32 Interactive Control is a utility 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.

- The GPIB configuration utility is integrated into the Windows 95 Device Manager. You use this utility to modify the configuration parameters of the NI-488.2M software.
- Hardware Diagnostic is a utility that you use to verify that the hardware is installed and functioning properly.
- Software Diagnostic is a utility that you use to verify that the NI-488.2M software has been installed properly.
- GPIB Information is a utility you can use to obtain information about your GPIB hardware and software, such as the version of the NI-488.2M software and the type of interface board you have installed.

16-bit Windows Support Files

- `gpib.dll` is the 16-bit Windows dynamic link library. When you run an existing NI-488.2 application for Windows in the Windows 95 environment, this file replaces the GPIB DLL that you used in the Windows 3 environment for Win16 applications.
- `gpib32ft.dll` is the 32-bit Windows dynamic link library that helps `gpib.dll` thunk 16-bit GPIB calls to 32-bit GPIB calls that address the standard 32-bit dynamic link library, `gpib-32.dll`.

Microsoft C/C++ Language Interface Files

- `readme.txt` is a documentation file that contains information about the C language interface.
- `decl-32.h` is a 32-bit include file. It contains NI-488 function and NI-488.2 routine prototypes and various predefined constants.
- `gpib-32.obj` is a 32-bit C language interface file. An application links with this file in order to access the 32-bit DLL.

Microsoft Visual Basic Language Interface Files

- `readme.txt` is a documentation file that contains information about the Visual Basic language interface.
- `niglobal.bas` is a Visual Basic global module that contains certain predefined constant declarations.
- `vbib-32.bas` is a Visual Basic source file with NI-488.2 routine and NI-488 function prototypes.

Sample Application Files

The NI-488.2M software includes nine sample applications along with source code for each language supported by the NI-488.2M software. For a detailed description of the sample application files, refer to Chapter 2, *Application Examples.*

How the NI-488.2M Software Works with Windows 95

The NI-488.2M software for Windows 95 includes a multi-layered device driver that consists of DLL pieces that run in user mode and VxD pieces that run in kernel mode. User applications access this device driver from user mode through `gpib-32.dll`, a 32-bit Windows 95 dynamic link library.

GPIB applications access the NI-488.2M software through `gpib-32.dll` as follows:

- A Win32 application can either link with the language interface (`gpib-32.obj`) or directly access the functions exported by the DLL.
- If you already have an existing Win16 application, use the 16-bit DLL (`gpib.dll`) to access the GPIB driver.

Figure 1-4 shows how you can use the NI-488.2M software with Windows 95 and your GPIB application programs.

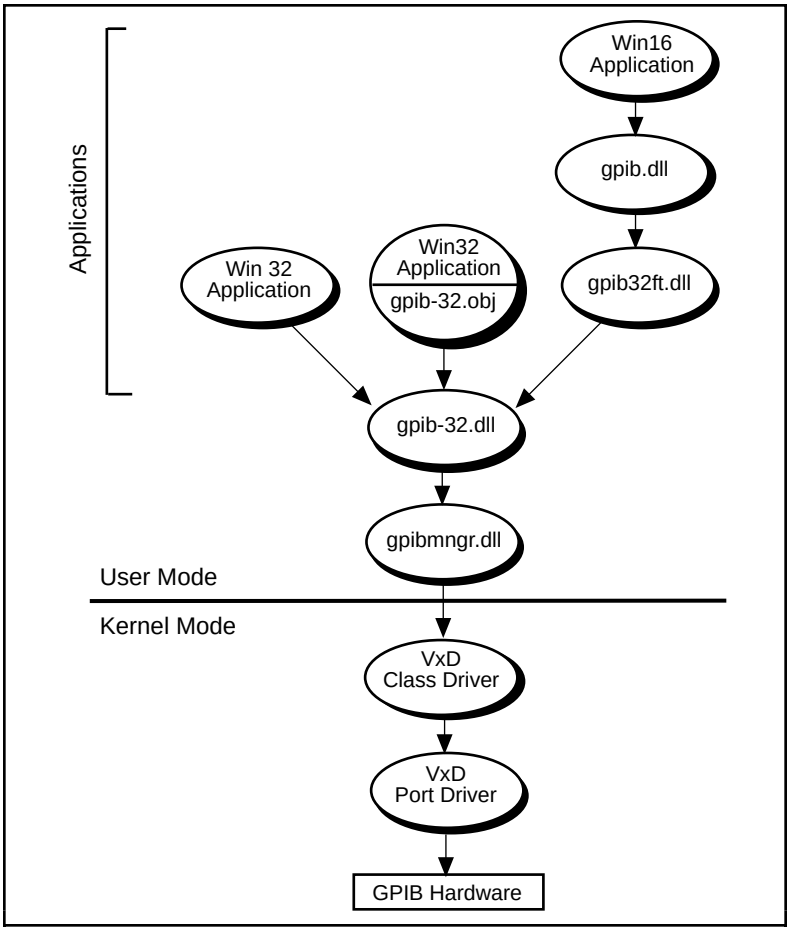


Figure 1-4. How the NI-488.2M Software Works with Windows 95

Uninstalling the Plug and Play GPIB Hardware

Before physically removing the Plug and Play GPIB hardware from the computer, you must remove the hardware information from the Windows 95 Device Manager.

To remove the hardware information from Windows 95, double-click the **System** icon in the **Control Panel**, which can be opened from the **Settings** selection of the **Start** menu. Select the **Device Manager** tab in the **System Properties** dialog box that appears, click the **View devices by type** button at the top of the **Device Manager** tab, and double-click on the **National Instruments GPIB Interfaces** icon.

To remove an interface, select it from the list of interfaces under **National Instruments GPIB Interfaces** as shown in Figure 1-5, and click the **Remove** button.

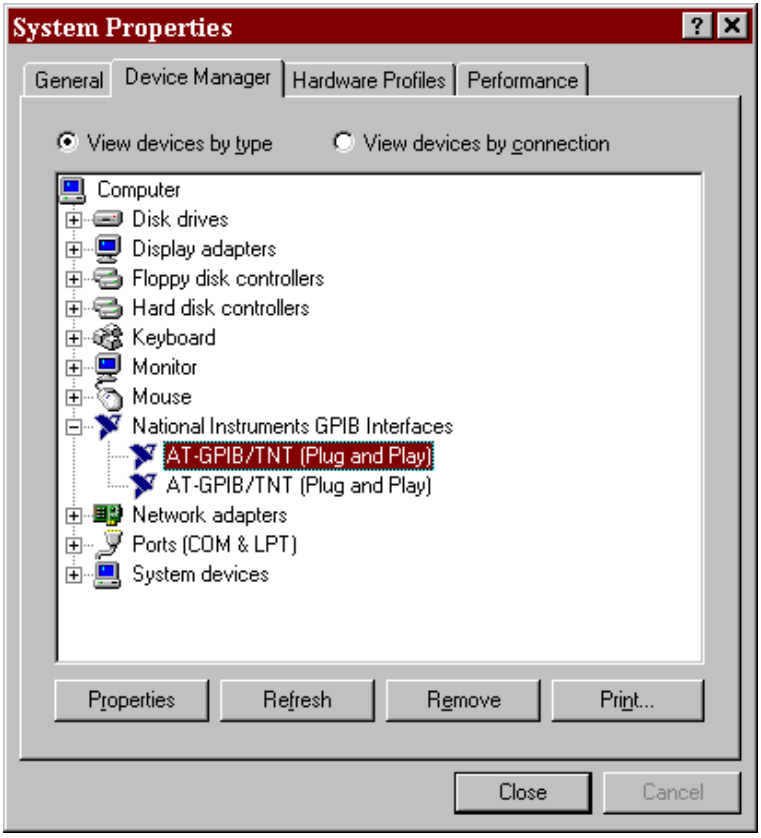


Figure 1-5. Selecting an Interface to Remove

After you remove the appropriate interface information from the Device Manager, you should physically remove the interface from your computer.

Uninstalling the Plug and Play GPIB Software

Before uninstalling the software, you should remove all GPIB interface information from the Windows 95 Device Manager, as described in the previous section. You do not need to shut down Windows 95 before uninstalling the software.

Complete the following steps to remove the Plug and Play GPIB software.

1. Run the **Add/Remove Programs** applet from the **Control Panel**, which can be opened from the **Settings** selection of the **Start** menu. A dialog box similar to the one in Figure 1-6 appears. This dialog box lists the software available for removal.

Figure 1-6. Add/Remove Programs Properties Dialog Box

2. Select the GPIB software you want to remove, and click the **Add/Remove...** button. The uninstall program runs and removes all folders, programs, VxDs, DLLs, and registry entries associated with the GPIB software. Figure 1-7 shows the results of a successful uninstallation.

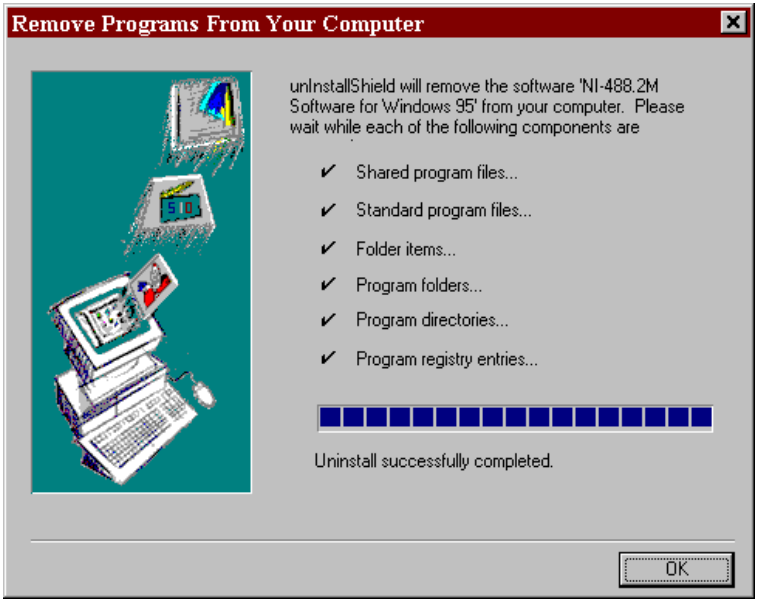


Figure 1-7. Uninstallation Results

If you have interfaces other than PCMCIA cards and you have not physically removed them from your computer, you should shut down Windows 95, power off your computer, and remove the interfaces now. You may remove PCMCIA cards without powering off your computer.

If you want to reinstall the hardware and software, refer to the getting started manual.

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-488.2M software, which includes the C 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-488.2M 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-488.2M 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.

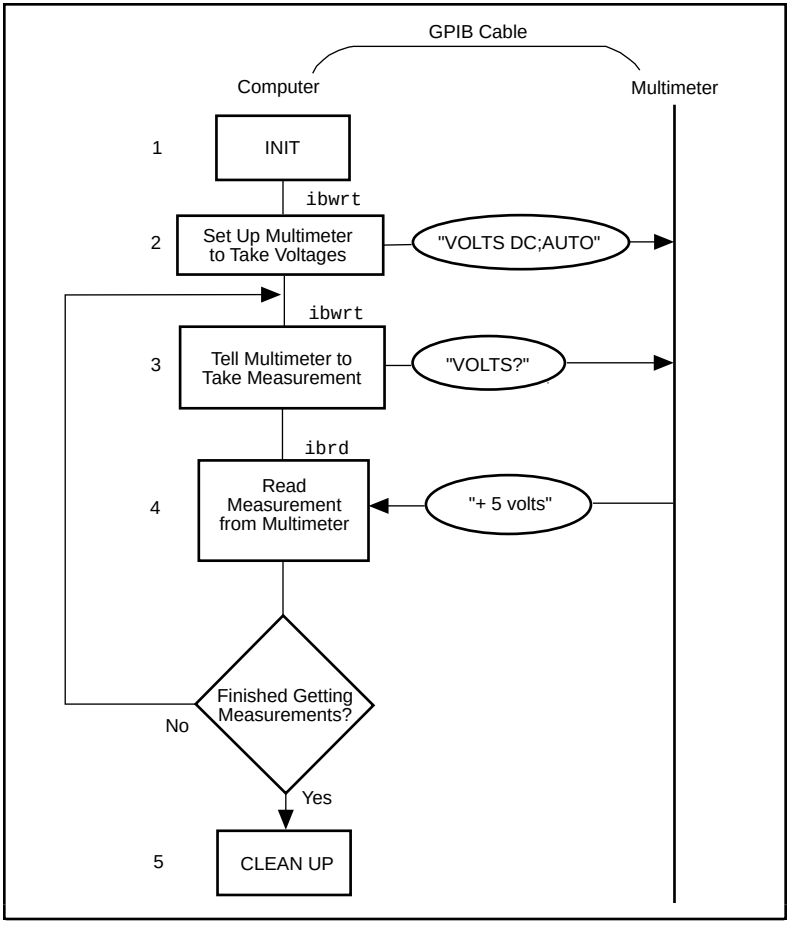


Figure 2-1. Program Flowchart for Example 1

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-488.2M 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.

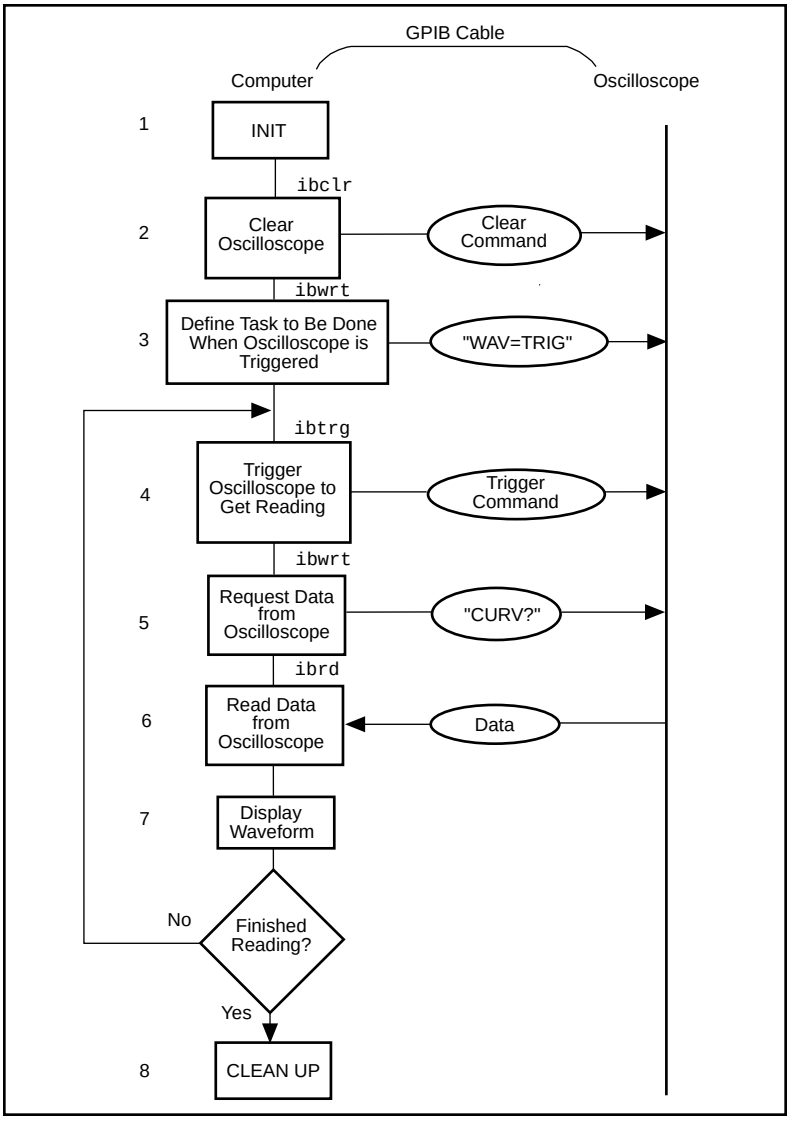


Figure 2-2. Program Flowchart for Example 2

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 lab 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 has the NI-488.2M software installed and is connected to a GPIB color printer. The biologist is familiar with the command set of her color printer, as described in the user manual provided by the manufacturer. 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.

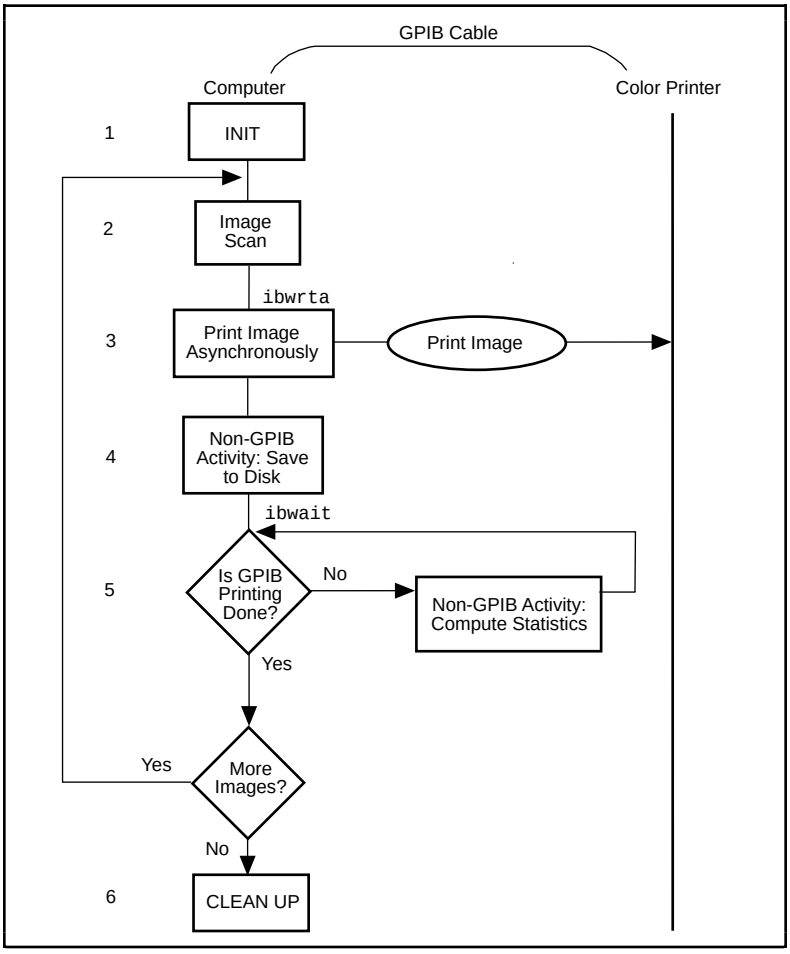


Figure 2-3. Program Flowchart for Example 3

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.

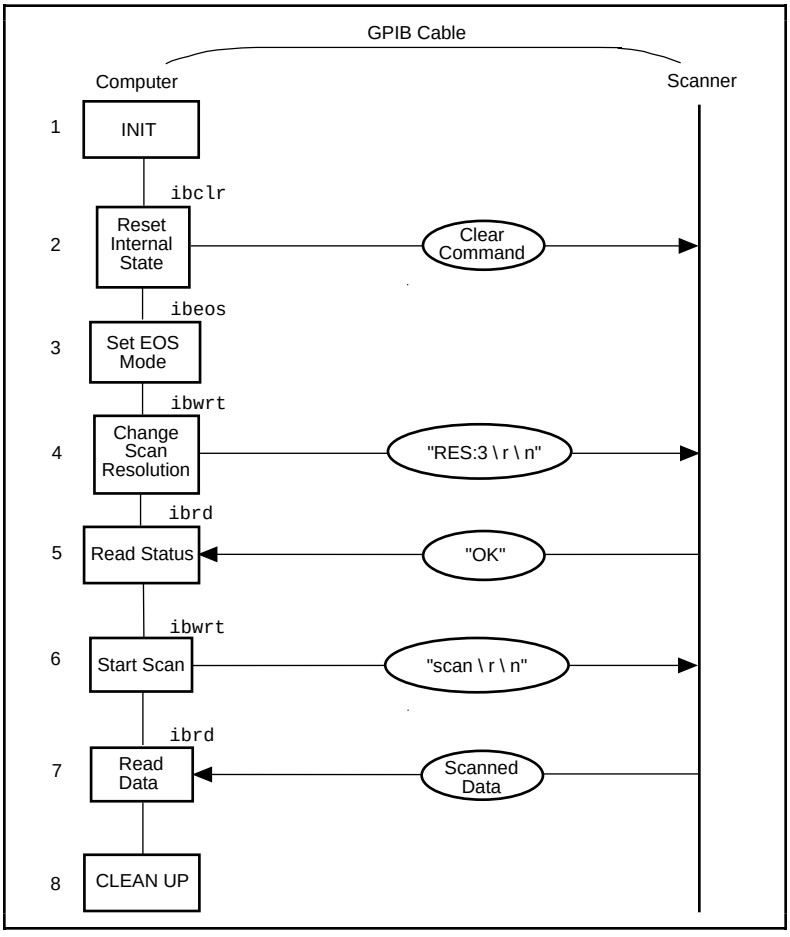


Figure 2-4. Program Flowchart for Example 4

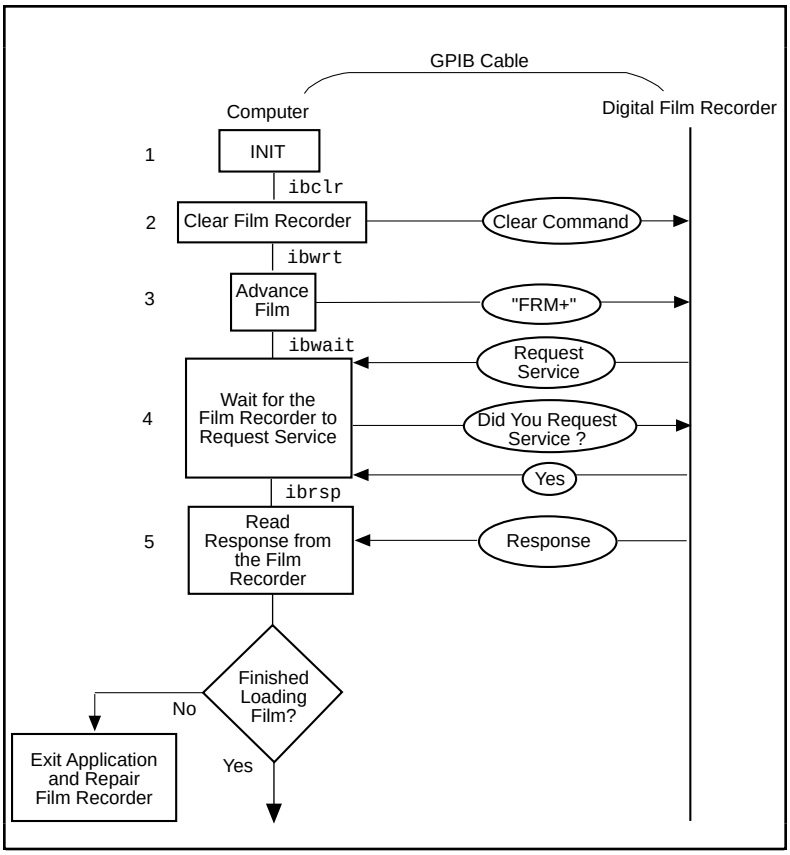
Example 5: Service Requests

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-488.2M software on the host computer and is familiar with the programming instructions for the film recorder, as described in the user manual provided by the manufacturer. 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.



(continues)

Figure 2-5. Program Flowchart for Example 5

(Continued)

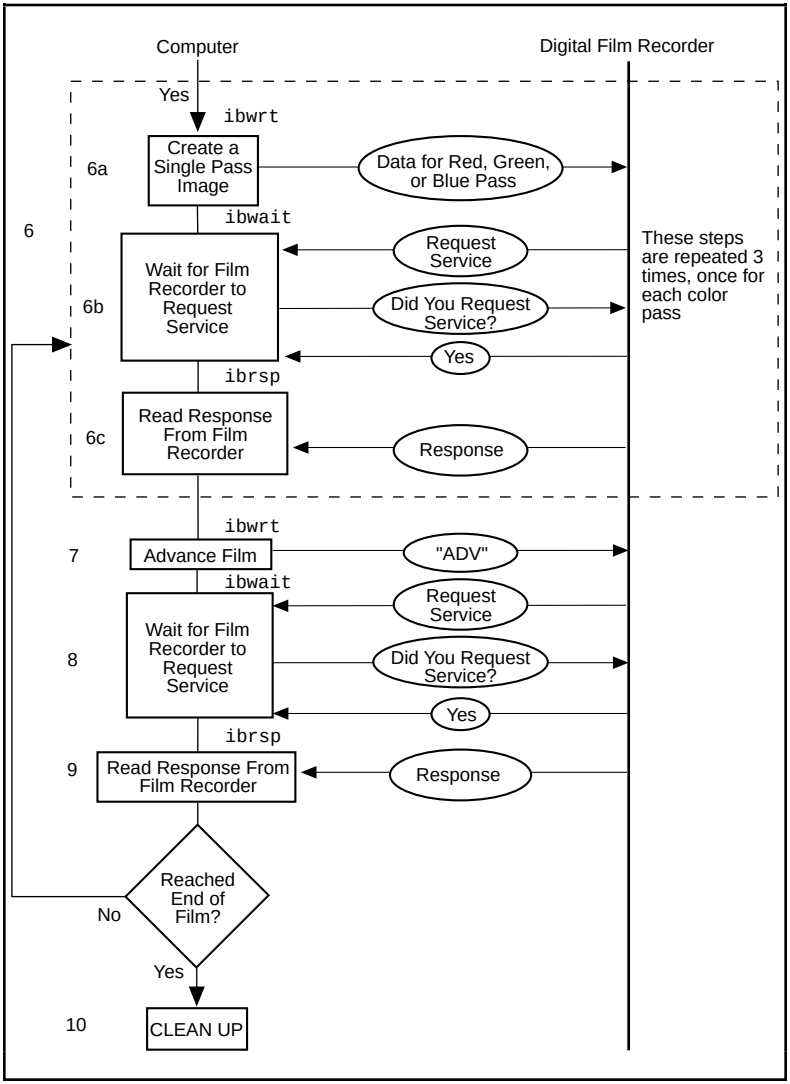


Figure 2-5. Program Flowchart for Example 5 (Continued)

Example 6: Basic Communication with IEEE 488.2-Compliant Devices

This example provides 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 find out 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 what GPIB addresses they have been set to. 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?"`) 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.

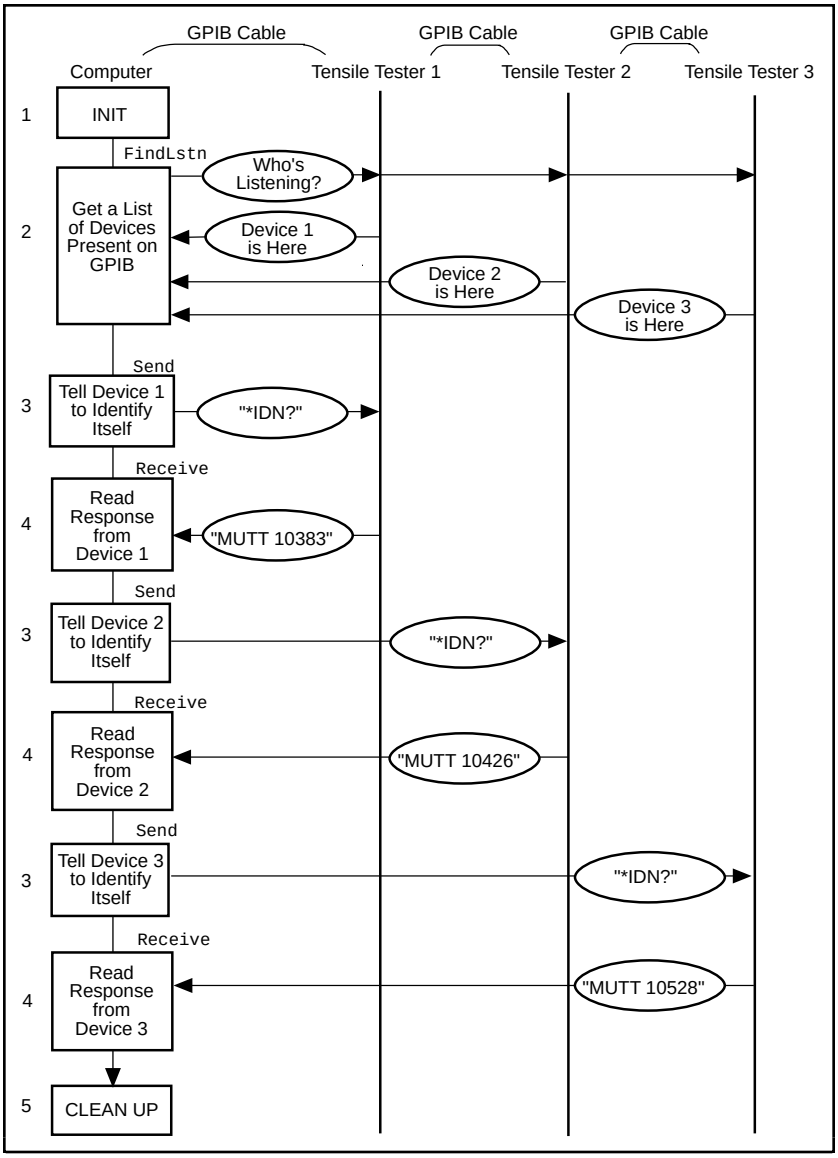


Figure 2-6. Program Flowchart for Example 6

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-488.2M 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.

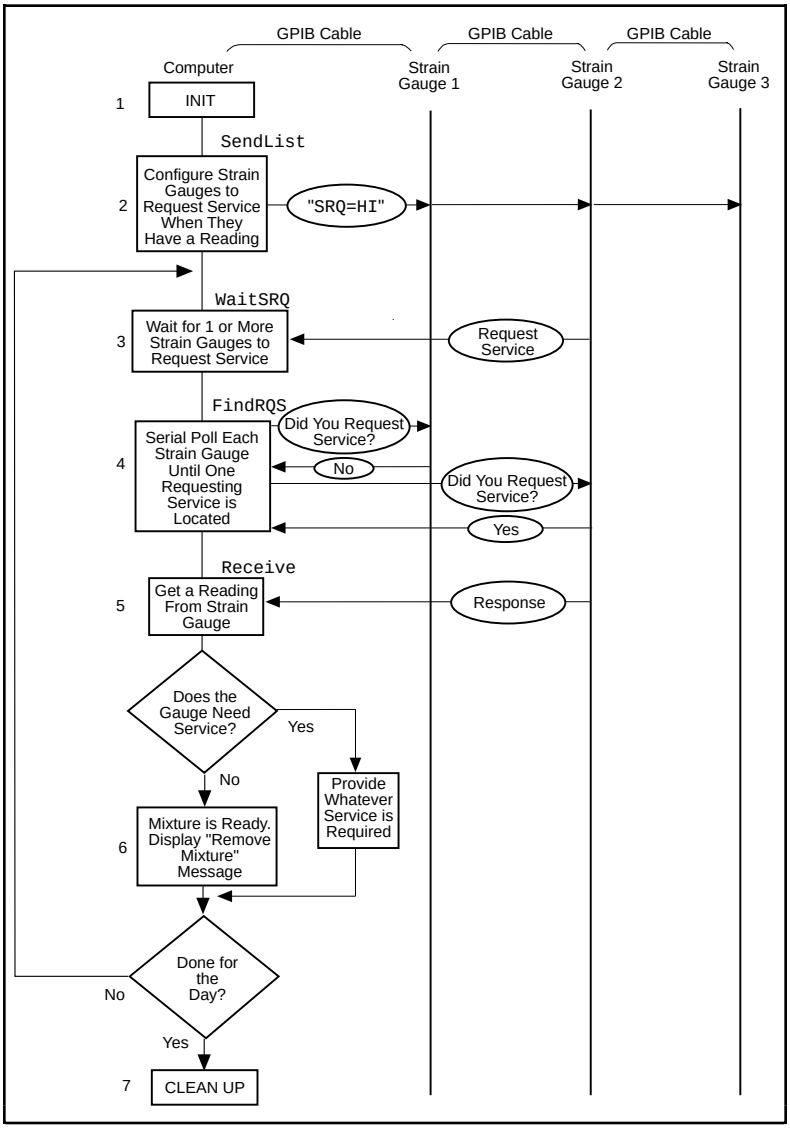


Figure 2-7. Program Flowchart for Example 7

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-488.2M 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.

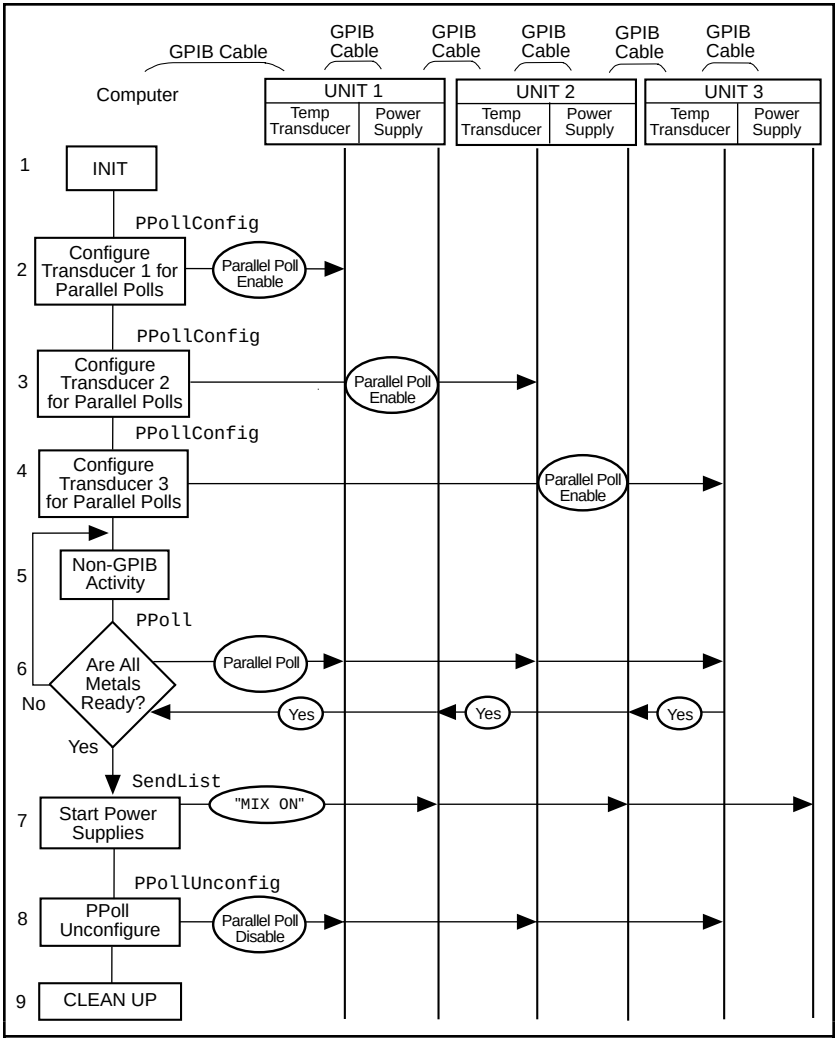


Figure 2-8. Program Flowchart for Example 8

Example 9: Non-Controller Example

This example illustrates how you can use the NI-488.2M 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.

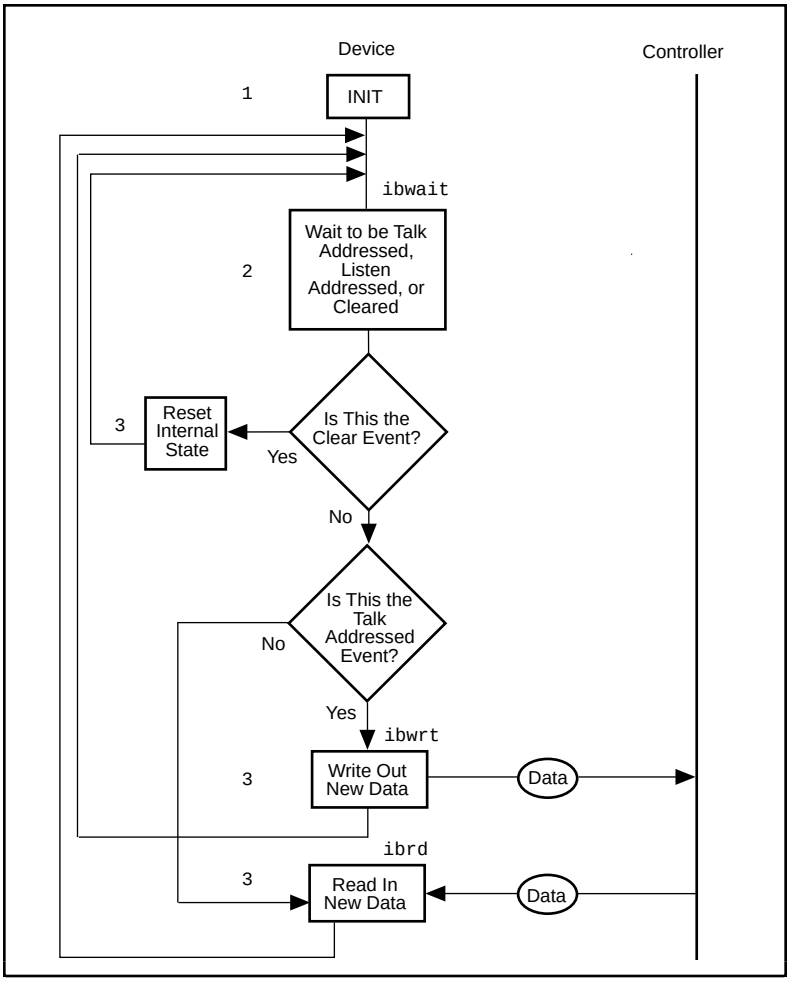


Figure 2-9. Program Flowchart for Example 9

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 How to Access gpib-32.dll

Applications can access the NI-488.2M dynamic link library (`gpib-32.dll`) either by using an NI-488.2M language interface or with direct access.

If you need to access the `gpib-32.dll` from a language other than Microsoft Visual C/C++ or Microsoft Visual Basic 4.0, you must directly access the `gpib-32.dll`. You can directly access the DLL from any programming environment that allows you to request addresses of variables and functions that a DLL exports. The `gpib-32.dll` exports pointers to each of the global variables:

- `user_ibsta` is a pointer to `ibsta`
- `user_iberr` is a pointer to `iberr`
- `user_ibcntl` is a pointer to `ibcntl`

The `gpib-32.dll` also exports pointers to all of the NI-488 and NI-488.2 calls. For example, it exports a pointer to the NI-488 `ibwrt` function. For a detailed example showing how to use direct access, refer to the sample program `dlldev.c` that came with your NI-488.2M software.

Choosing Between NI-488 Functions and NI-488.2 Routines

Your distribution disk contains two distinct sets of subroutines to meet your application needs. Both of these sets, the NI-488 functions and the NI-488.2 routines, are compatible across computer platforms and operating systems, so you can port programs to other platforms with little or no source code modification. 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. Regardless of which option you choose, the driver automatically addresses and performs other bus management operations necessary for device communication.

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 6, *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-488.2M routines only.

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

When your system includes a board that must access multiple devices, 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

Checking Status with Global Variables

Each NI-488 function and NI-488.2 routine updates four 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.

Status Word – ibsta

All functions update a global status word, `ibsta`, which contains information about the state of the GPIB and the GPIB hardware. The value stored in `ibsta` is the return value of all of the NI-488 functions except `ibfind` and `ibdev`. You can test for the conditions reported in `ibsta` and use that information to make decisions about continued processing. If you check for possible errors after each call, debugging your application is much easier.

`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 (brd) and NI-488.2 calls, or all (dev, brd).

Table 3-1 shows the condition that each bit position represents, the bit mnemonics, and the type of calls for which the 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 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
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 file included on your distribution disk contains 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 in*** `ibsta`***, 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` and `ibcntl` are 32-bit integers. On some systems, like MS-DOS, `ibcnt` is a 16-bit integer, and `ibcntl` is a 32-bit integer. For cross-platform compatibility, all applications should use `ibcntl`. 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 Win32 Interactive Control to Communicate with Devices

Before you begin writing your application program, you might want to use the Win32 Interactive Control utility. With Win32 Interactive Control, you communicate with your instruments from the keyboard rather than from an application program. You can use Win32

Interactive Control to learn to communicate with your instruments using the NI-488 functions or NI-488.2 routines. For specific device communication instructions, refer to the user manual that came with your instrument. For information about using Win32 Interactive Control and for detailed examples, refer to Chapter 5, *Win32 Interactive Control Utility*.

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-488.2M software includes the source code for this example written in C (`devsamp.c`) and the source code for this example written to use direct entry to access the `gpib-32.dll` (`dlldev.c`).

The NI-488.2M 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 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:

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

  then your application invokes it 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 NI-488 functions. The flowchart is for device-level calls.

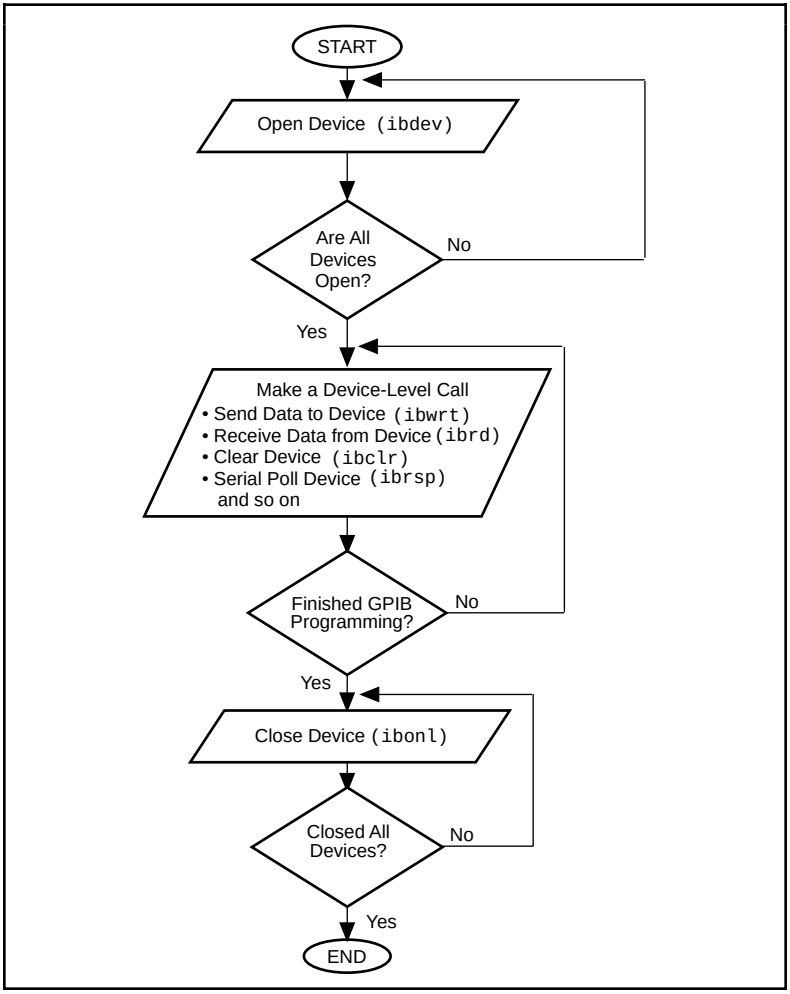


Figure 3-1. General Program Shell Using NI-488 Device Functions

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). The last argument represents the number of bytes to be sent.

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-488.2M software includes the source code for this example written in C (`samp4882.c`) and the code for this example written to use direct entry to access the `gpib-32.dll` (`dll4882.c`).

The NI-488.2M 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 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:

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

  Then your application invokes it 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.

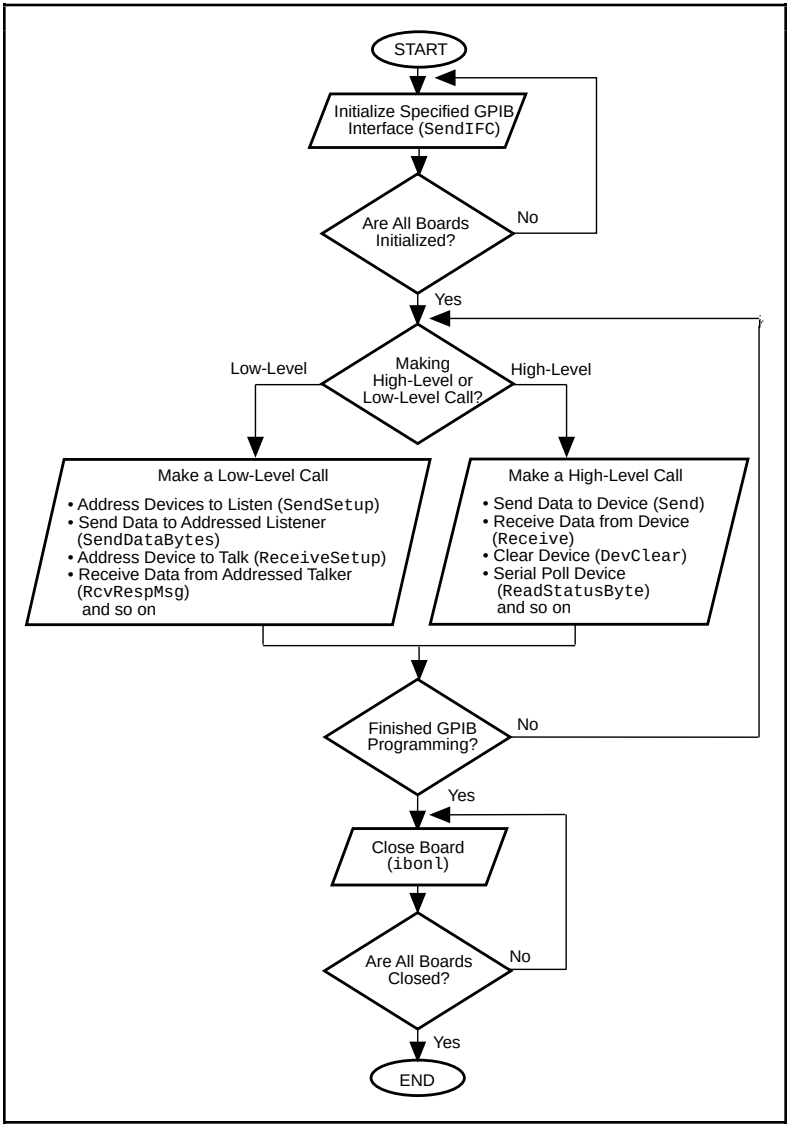


Figure 3-2. General Program Shell Using NI-488.2 Routines

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. (`AsciiToFloat` is a function that takes a null-terminated string as input and outputs the floating point number it represents.)

```
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 GPIB Win32 Application

The following sections describe how to compile, link, and run your Win32 GPIB application.

Microsoft Visual C/C++

Before you compile your Win32 C application, make sure that the following line is included at the beginning of your program:

```
#include “decl-32.h”
```

After you have written your C application program, you must compile the application program using Microsoft Visual C/C++ (version 2.0 or higher). Next, link the application with the C language interface,

gpib-32.obj. To compile and link a Win32 console application named cprog in a DOS shell, type the following on the command line:

```
cl cprog.c gpib-32.obj
```

To run your application from the Windows environment, choose the **Run...** option from the **Start** menu. Enter the name of the compiled program in the dialog box that pops up. To run your application from a DOS shell, type the name of your compiled program on the DOS command line.

Visual Basic

For Win32 applications, use Microsoft Visual Basic (version 4.0 or higher). Before you run your Visual Basic application, include the files niglobal.bas and vbib-32.bas in your application project file.

To run your application from the Visual Basic environment, choose the **Start** option from the **Run** menu to execute your program.

Direct Entry with C

Before you compile your Win32 C application, make sure that the following lines are included at the beginning of your application:

```
#ifdef __cplusplus
extern "C"{
#endif
#include "decl-32.h"
/* Global variable for the handle to the loaded
gpib-32.dll. */
HINSTANCE Gpib32Lib = NULL;
/* Pointers to NI-488.2 global status variables */
int *Pibsta;
int *Piberr;
long *Pibcntl;
#ifdef __cplusplus
}
#endif
```

In addition to pointers to the status variables and a handle to the loaded gpib-32.dll, you must define the direct entry prototypes for the functions you use in your application. The prototypes for each function exported by gpib-32.dll can be found in the *NI-488.2M Function Reference Manual for Win32*. The NI-488.2M direct entry sample programs illustrate how to use direct entry to access gpib-32.dll. LoadLibrary and GetProcAddress are used to load the

gpib-32.dll and get pointers to its exported functions. For more information on direct entry, refer to the Win32 SDK (Software Development Kit) online help.

In your Win32 application, you first need to load gpib-32.dll. The following code fragment illustrates how to call the LoadLibrary function and check for an error:

```
Gpib32Lib=LoadLibrary("GPIB-32.DLL");
if (Gpib32Lib == NULL) {
   return FALSE;
}
```

Next, your Win32 application needs to use GetProcAddress. The following code fragment illustrates how to get the addresses of the pointers to the status variables and any functions it needs to use:

```
Pibsta = (int *) GetProcAddress(Gpib32Lib,
       (LPCSTR)”user_ibsta”);
Piberr = (int *) GetProcAddress(Gpib32Lib,
       (LPCSTR)”user_iberr”);
Pibcntl = (long *) GetProcAddress(Gpib32Lib,
       (LPCSTR)”user_ibcnt”);
Pibdev = (int (__stdcall *)(int, int, int, int, int,
       int)) GetProcAddress(Gpib32Lib,
       (LPCSTR)"ibdev");
Pibonl = (int (__stdcall *)(int, int))
       GetProcAddress(Gpib32Lib, (LPCSTR)"ibonl");
```

If GetProcAddress fails, it returns a NULL pointer. The following code fragment illustrates how to verify that none of the calls to GetProcAddress failed:

```
if ((Pibsta  == NULL) ||
    (Piberr  == NULL) ||
    (Pibcntl == NULL) ||
    (Pibdev  == NULL) ||
    (Pibonl  == NULL)) {
     // ERROR!
   }
```

Your Win32 application dereferences the pointer to access either the status variables or function. The following code illustrates how to call a function and access the status variable from within your application:

```
dvm = (*Pibdev) (0, 1, 0, T10s, 1, 0);
if (*Pibsta & ERR) {
   printf("Call failed");
}
```

Before exiting your application, you need to free `gpib-32.dll` with the following command:

```
FreeLibrary(Gpib32Lib);
```

For more information on direct entry, refer to the Win32 SDK (Software Development Kit) online help.

Microsoft Visual C/C++

After you have written your Win32 application, you must compile the application using Microsoft Visual C/C++ (version 2.0 or higher). To compile and link a Win32 console application named `cprog` in a DOS shell, type the following on the command line:

```
cl cprog.c
```

To run your application from the Windows environment, choose the **Run...** option from the **Start** menu. Enter the path and name of the compiled program in the dialog box that appears. To run your application from a DOS shell, type the name of your compiled program on the DOS command line.

Borland C/C++

After you have written your Win32 Borland C/C++ (version 4.0 or higher) application, compile it using the `-w32` option to create a console application. From the command line in a DOS shell, type the following command to compile and link a Win32 application named `cprog`:

```
bcc32 -w32 cprog.c
```

To run your application from the Windows environment, choose the **Run...** option from the **Start** menu. Enter the name of the compiled program in the dialog box that appears. To run your application from a DOS shell, type the name of your compiled program on the DOS command line.

Running Existing Win16 GPIB Applications

You can run existing Win16 GPIB applications under Windows 95 by using the pair of 16-to-32 bit thunking DLLs, `gpib.dll` and `gpib32ft.dll`, which are included with your NI-488.2M software.

To run 16-bit Windows GPIB applications, the system uses the special GPIB dynamic link library, `gpib.dll`. When you install the NI-488.2M software, `gpib.dll` and `gpib32ft.dll` are copied into the Windows System directory. These DLLs are automatically accessed whenever you execute a Win16 GPIB application.

Chapter 4

Debugging Your Application

This chapter describes several ways to debug your application program.

Running GPIB Information

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

Run GPIB Information with no 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. GPIB Information also displays information about each GPIB interface board installed in your system, including the name of the board, the type of Controller chip it uses, the hardware settings, the type of functions that the board can use, and whether or not the board can use the HS488 high-speed communication protocol. The typical GPIB Information output is as follows:

```
GPIB Information (Oct 9 1995)
Copyright 1995 National Instruments Corp.  All rights
reserved.
Software Information:
  The NI-488.2M Software for Windows 95 is loaded.
  You are running Version 1.0.
  It supports both the NI-488 functions and the
    NI-488.2 routines.
  It supports the HS488 high-speed protocol.

Hardware Information:
  GPIB0: 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.
```

Debugging with the Global Status Variables

After each function call to your NI-488.2M 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 can help you interpret the state of the driver.

Debugging with Win32 Interactive Control

If your application does not automatically check for and display errors, you can locate an error by using the Win32 Interactive Control utility. Simply issue the same functions or routines, one at a time as they appear in your application program. Because Win32 Interactive Control 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 Win32 Interactive Control , refer to the online help.

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 can help you interpret the state of the driver.

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	System 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
EDMA	8	DMA error
EOIP	10	Asynchronous I/O in progress
ECAP	11	No capability for operation
EFSO	12	File system error
EBUS	14	GPIB bus error
ESTB	15	Serial poll status byte queue overflow
ESRQ	16	SRQ stuck in ON position
ETAB	20	Table problem

Troubleshooting EDVR Error Conditions

In some cases, calls to NI-488 functions or NI-488.2 routines may return with the ERR bit set in `ibsta` and the value EDVR in `iberr`. The value stored in `ibcntl` is useful in troubleshooting the error condition.

EDVR Error with ibcntl Set to 0xE028002C

If a call is made with a board number that is within the range of allowed board numbers (typically 0 to 3), but which has not been assigned to a GPIB interface, an EDVR error condition occurs with `ibcntl` set to 0xE028002C. You can assign a board number to a GPIB interface by configuring the NI-488.2M software and selecting an interface name. Refer to the getting started manual for information on how to configure the NI-488.2M software.

EDVR Error with ibcntl Set to 0xE0140025

If a call is made with a board number that is not within the range of allowed board numbers (typically 0 to 3), an EDVR error condition occurs with `ibcntl` set to 0xE0140025.

EDVR Error with ibcntl Set to 0xE0140035

If a call is made with a device number that is not within the range of allowed device numbers (typically 1 to 32), an EDVR error condition occurs with `ibcntl` set to 0xE0140035.

EDVR Error with ibcntl Set to 0xE0320029

If a call is made with a board number that is assigned to a GPIB interface that is unusable because of a resource conflict, an EDVR error condition occurs with `ibcntl` set to 0xE0320029. Refer to the troubleshooting instructions in the getting started manual. This error is also returned if you remove a PCMCIA-GPIB or PCMCIA-GPIB+ while the driver is accessing it.

Configuration Errors

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 either permanently reconfigure the driver using the NI-488.2M software configuration utility, which is integrated with the Windows 95 Device Manager, or temporarily reconfigure the driver while your application is running using the `ibconfig` function.

☞ **Note:** ***National Instruments recommends using*** `ibconfig` ***to modify the NI-488.2M driver configuration dynamically.***

If your program uses dynamic configuration, it will always work regardless of the previous configuration of the driver. Refer to the description of `ibconfig` in the *NI-488.2M Function Reference Manual for Win32* for more information.

Timing Errors

If your application fails, but the same calls issued in the Win32 interactive control utility 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

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. However, some devices require GPIB addressing before any GPIB activity. Therefore, you might need to configure your NI-488.2M driver to perform repeat addressing if your device does not remain in its currently addressed state. Refer to Chapter 7, *GPIB Configuration Utility*, or to the description of `ibconfig` (option `IbcREADDR`) in the *NI-488.2M Function Reference Manual for Win32* 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-488.2M 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

How can I determine which type of GPIB hardware I have installed?

Run the GPIB Information utility. To run the utility, select the **GPIB Information** item under **Start»Programs»NI-488.2M Software for Windows 95**. GPIB Information returns information about the GPIB boards currently configured for use in your system.

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

Run the GPIB Information utility. To run the utility, select the **GPIB Information** item under **Start»Programs»NI-488.2M Software for Windows 95**. GPIB Information returns information about the version of the NI-488.2M software currently installed.

How can I determine if my GPIB hardware and software are correctly installed?

Refer to the getting started manual for instructions on running the hardware and software diagnostic tests.

When should I use the Win32 Interactive Control utility?

You can use the Win32 Interactive Control utility to test and verify instrument communication, troubleshoot problems, and develop your application program. For more information, refer to Chapter 5, *Win32 Interactive Control Utility*.

How do I use an NI-488.2M language interface?

For information about using NI-488.2M language interfaces, refer to Chapter 3, *Developing Your Application*.

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.

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 `iberr`. For more information about global status variables, refer to Chapter 3, *Developing Your Application.*

What information should I have before I call National Instruments?

When you call National Instruments, you should have the results of the hardware and software diagnostic tests along with the output from the GPIB Information utility. Also, make sure you have filled out the technical support form in Appendix C, *Customer Communication.*

Chapter 5

Win32 Interactive Control Utility

This chapter introduces you to Win32 Interactive Control, the interactive control utility that you can use to communicate with GPIB devices interactively.

Overview

With the Win32 Interactive Control utility, 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 Win32 Interactive Control to practice communication with the instrument, troubleshoot problems, and develop your application program.

One way Win32 Interactive Control 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 Win32 Interactive Control 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, Win32 Interactive Control prompts you for the necessary parameters.

1. To run Win32 Interactive Control, select the **Win32 Interactive Control** item under **Start»Programs»NI-488.2M Software for Windows 95**. Your screen should appear as follows:

```
National Instruments
IEEE 488 Interface Bus Interactive Control Program
(IBIC)
Copyright 1993 National Instruments Corp. Version 3.0
(Win32)
Version Date: Oct 6 1995 Version Time: 09:42:25
All rights reserved

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

:
```

2. Use `ibdev` to find the device name which is assigned to your device in the GPIB Configuration Utility. 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 user manual for the command bytes that work with your instrument.

```
ud0: ibwrt
    enter string: "F3R7T3"
[0100] (cmpl)
count: 6
```

or

```
ud0: ibwrt "F3R7T3"
[0100] (cmpl)
count: 6
```

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 Win32 Interactive Control utility by entering `q` at the prompt.

Win32 Interactive Control Syntax

When you enter commands in Win32 Interactive Control, 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 as an ASCII character sequence, octal bytes, hex bytes, or special symbols.

ASCII character sequence–You must enclose the entire sequence in quotation marks.

Octal bytes–You must use a backslash character followed by the octal value. For example, octal 40 is represented by `\40`.

Hex bytes–You must use a backslash character and an `x` followed by the hex value. For example, hex 40 is represented by `\x40`.

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.

Win32 Interactive Control Syntax for NI-488 Functions

Tables 5-1 and 5-2 summarize the syntax of NI-488 functions in Win32 Interactive Control. `v` represents a number that you input. `string` represents a string that you input. For more information about the function parameters, use the online help feature.

Table 5-1. Syntax for Device-Level NI-488 Functions in Win32 Interactive Control

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 Win32 Interactive Control

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
`ibfind udname`	Return unit descriptor where `udname` is the symbolic name of 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
`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

Win32 Interactive Control Syntax for NI-488.2 Routines

Table 5-3 summarizes the syntax of NI-488.2 routines in Win32 Interactive Control. `v` represents a number that you input 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 built-in online help feature or refer to the *NI-488.2M Function Reference Manual for Win32*.

Table 5-3. Syntax for NI-488.2 Routines in Win32 Interactive Control

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 addrlist 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 address`	Unconfigure device for parallel poll
`RcvRespMsg address string v`	Receive response message
`ReadStatusByte address`	Serial poll a device
`Receive address string 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 addrlist string v`	Send data bytes
`SendIFC`	Send interface clear
`SendList addrlist string v`	Send data to multiple devices
`SendLLO`	Put devices in local lockout

(continues)

Table 5-3. Syntax for NI-488.2 Routines in Win32 Interactive Control (Continued)

Routine Syntax	Description
`SendSetup addrlist`	Send setup
`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 Win32 Interactive Control, 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, Win32 Interactive Control 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, Win32 Interactive Control 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 Win32 Interactive Control 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-488.2M Function Reference Manual for Win32* 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. `gpib0` is the default name found in the driver. For more information about valid names, refer to Chapter 7, *GPIB 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, Win32 Interactive Control 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 the board has no driver installed. The following example illustrates an EDVR error.

```
:
ibdev 4 6 0x67 7 1 0
id = -1
[8000] (err)
error: EDVR (0xe0140025)

:
```

- ENEB—The board index entered refers to a known board (such as 0), but the driver cannot find the board. In this case, use the configuration utility as described in Chapter 7, *GPIB Configuration Utility*, and make sure you have correctly associated a logical name (`gpib0`, `gpib1`, and so on) with each physical interface.
- 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, the 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: 30
```

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 called `dev1` you enter the following string at the `dev1` 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 Win32 Interactive Control

Set 488.2

You must use the set command before you can use NI-488.2 routines in Win32 Interactive Control. 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).

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 all 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 Win32 Interactive Control utility 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 Win32 Interactive Control.

Table 5-4. Auxiliary Functions in Win32 Interactive Control

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 device or board.
`Set 488.2 v`	Enter 488.2 mode for board v
`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 OFF display.
`+`	Turn ON display.
`n* function`	Execute function `n` times where `function` represents the correct Win32 Interactive Control function syntax.
`n* !`	Execute previous function `n` times.
`$ filename`	Execute indirect file where `filename` is the pathname of a file that contains Win32 Interactive Control functions to be executed.
`print string`	Display string on screen where `string` is an ASCII character sequence, octal bytes, hex bytes, or special symbols.
`e`	Exit or quit.
`q`	Exit or quit.

Set (udname or 488.2)

You can use the `set` command to select 488.2 mode or to communicate with a different device or board.

The following example shows how to enter 488.2 mode. The `488.2 (0)` prompt indicates that you are in NI-488.2 mode on board `0`.

```
: set 488.2 0

488.2 (0):
```

The next 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 launches the windows help file for the Win32 Interactive Control Utility. 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 5 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 Win32 Interactive Control functions listed in that file, in sequence, as if they were entered in that order from the keyboard. The following example executes the Win32 Interactive Control 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.

Chapter 6

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-488.2M Function Reference Manual for Win32*.

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-488.2M 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-488.2M 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:

- A 7-bit or 8-bit 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-488.2M driver automatically asserts the GPIB EOI line when the EOS byte is written to the GPIB. 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 an `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-488.2M driver terminates `ibrd`, `ibrda`, and `ibrdf` calls when the EOS byte is detected on the GPIB or 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 in. Use the `IbcEndBitIs` Normal option to configure the software to report only the END bit in `ibsta` when the GPIB EOI line is asserted. By default, the NI-488.2M 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 GPIB Information utility. If you attempt to enable HS488 on a GPIB board that does not have the TNT4882C hardware, 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-488.2M software automatically sends the HS488 configuration message to devices. If you enabled the HS488 protocol in the GPIB Configuration Utility, the NI-488.2M 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-488.2M 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 properly in the GPIB Configuration Utility, 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-488.2M Function Reference Manual for Win32.*

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-488.2M 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 GPIB 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-488.2M 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-488.2M software in most non-Controller situations. These situations are known as Talker/Listener applications because the interface board is not the GPIB 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.

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. The autopolling procedure occurs as follows:

1. To enable autopolling, use the GPIB Configuration Utility 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 an `ibwait` for RQS is made. When `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-488.2M software can perform autopolling after any device-level NI-488 call as long as no GPIB I/O is currently in progress. In this case, an automatic serial poll can occur even when your application is not making any calls to the NI-488.2M 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.

☞ **Note:** ***The NI-488.2M software for Windows 95 does not function properly if interrupts are disabled.***

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-32.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-488.2M 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 only serial polls 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 devices on the list. 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 find the first device that 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");
      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 with a single call.

```
void GetAllSerialPollResponses ( Addr4882_t
AddrList[], short ResponseList[] )
{
   int WaitResult;
   WaitSRQ (0, &WaitResult);
   if ( WaitResult ) {
      printf ( "SRQ is asserted.\n" );
      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 parallel polling is that a single parallel poll 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. The value of the individual status bit (`ist`) determines the parallel poll response.

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.

The following call configures the device to respond to the poll on DIO7 and to assert the line in the case when its `ist` is 0. Pass the binary bit pattern, 0110 0110 or hex 66, to `ibppc`.

```
#include "decl-32.h"
char ppr;
dev = ibdev(0,3,0,T3s,1,0);
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-32.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 7

GPIB Configuration Utility

This chapter contains a description of the GPIB configuration utility you can use to configure your NI-488.2M software.

Overview

The GPIB configuration utility is integrated into the Windows 95 Device Manager. You can use it to view or modify the configuration of your GPIB interface boards. You can also use it to view or modify the GPIB device templates, which provide compatibility with older applications. The online help includes all of the information that you need to properly configure the NI-488.2M software.

In most cases, you should use the GPIB configuration utility only to change the hardware configuration of your GPIB interface boards. To change the GPIB characteristics of your boards and the configuration of the device templates, use the `ibconfig` function in your application program. If your application program uses `ibconfig` whenever it needs to modify a configuration option, it is able to run on any computer with the appropriate NI-488.2M software, regardless of the configuration of that computer.

Configure the NI-488.2M Software

You do not need to configure the NI-488.2M software unless you are using more than one GPIB interface in your system. If you are using more than one interface, you should configure the NI-488.2M software to associate a logical name (`gpib0`, `gpib1`, and so on) with each physical GPIB interface.

☞ **Note:** ***GPIB Analyzer software settings are available through the GPIB Analyzer application.***

To configure the NI-488.2M software, follow these steps:

1. Double-click the **System** icon in the **Control Panel**, which can be opened from the **Settings** selection of the **Start** menu.
2. Select the **Device Manager** tab in the **System Properties** dialog box that appears.
3. Click the **View devices by type** radio button at the top of the **Device Manager** tab, and double-click the **National Instruments GPIB Interfaces** icon.
4. Double-click on the particular interface type you want to configure in the list of installed interfaces immediately below **National Instruments GPIB Interfaces**. The **Resources** tab provides information about the hardware resources assigned to the GPIB interface, and the **NI-488.2M Settings** tab provides information about the software configuration for the GPIB interface.
5. Use the **Interface Name** drop-down box to select a logical name (`GPIB0`, `GPIB1`, and so on) for the GPIB interface. Repeat this process for each interface you need to configure. Figure 7-1 shows the **NI-488.2M Settings** tab for an AT-GPIB/TNT (PnP).

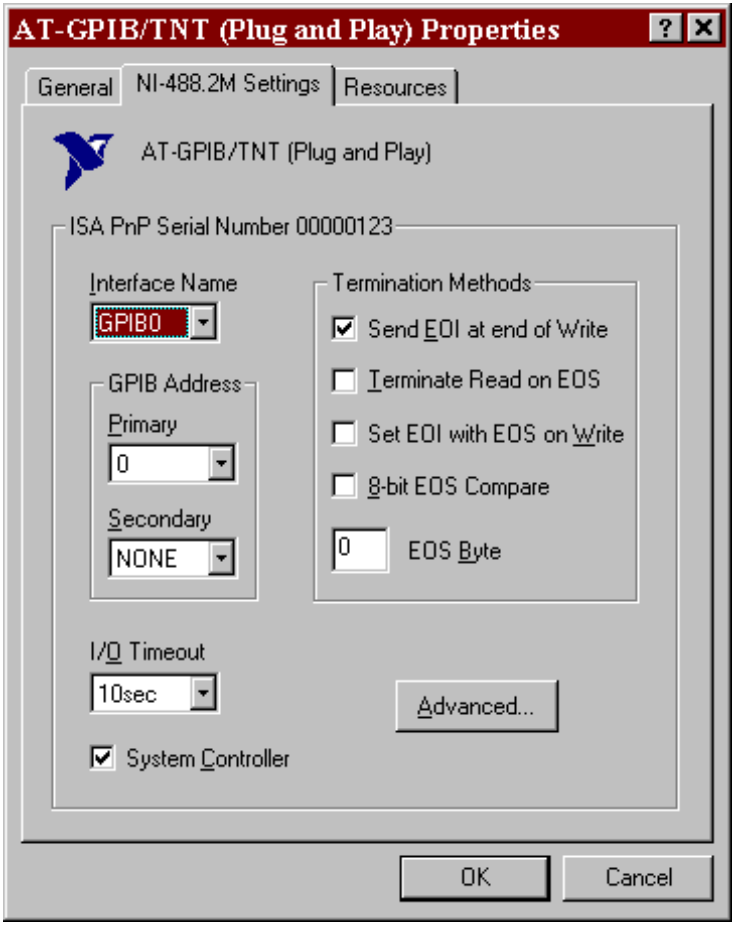


Figure 7-1. NI-488.2M Settings Tab for the AT-GPIB/TNT (PnP)

If you want to examine or modify the logical device templates for the GPIB software, select the **National Instruments GPIB Interfaces** icon from the **Device Manager** tab, and click the **Properties** button. Select the **Device Templates** tab to view the logical device templates, as shown in Figure 7-2.

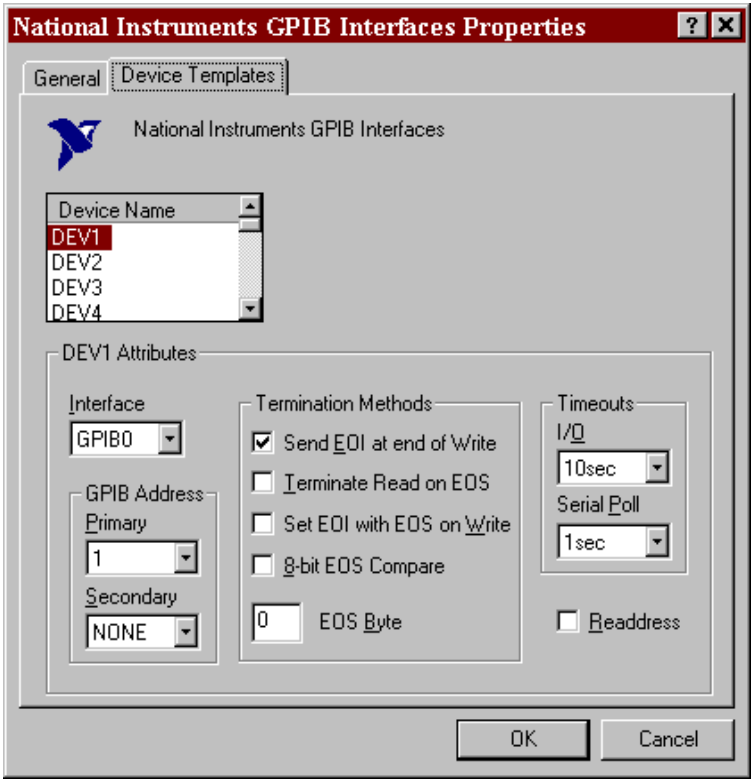


Figure 7-2. Device Templates Tab for the Logical Device Templates

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 shows the status word 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
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 and indicates that the device is requesting service. RQS is set whenever bit 6 is asserted in the serial poll status byte of the device. The serial poll that obtains the status byte can be the result of a call to `ibrsp`, or the poll might be automatic if automatic serial polling is enabled. Do not issue an `ibwait` on RQS for a device that does not respond to serial polls. RQS is cleared when an `ibrsp` reads the serial poll status byte that caused the RQS.

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.

Error Mnemonic	iberr Value	Meaning
EDVR	0	System 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
EDMA	8	DMA error
EOIP	10	Asynchronous I/O in progress
ECAP	11	No capability for operation
EFSO	12	File system error
EBUS	14	GPIB bus error
ESTB	15	Serial poll status byte queue overflow
ESRQ	16	SRQ stuck in ON position
ETAB	20	Table problem

EDVR (0)

EDVR is returned when the board or device name passed to `ibfind`, or the board index passed to `ibdev`, cannot be accessed. The global variable `ibcntl` contains the system error code 2, *File Not Found.* This error occurs when you try to access a board or device that is not installed or configured properly.

EDVR is also returned if an invalid unit descriptor is passed to any NI-488 function call.

Solutions

- Use `ibdev` to open a device without specifying its symbolic name.
- Use only device or board names that are configured in the GPIB configuration utility as parameters to the `ibfind` function.
- Use the unit descriptor returned from `ibdev` or `ibfind` as the first parameter in subsequent NI-488 functions. Examine the variable before the failing function to make sure its value has not been corrupted.
- Refer to the *Troubleshooting EDVR Error Conditions* section in Chapter 4, *Debugging Your Application*, for more information.

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: `ibcmd`, `ibcmda`, `ibln`, and `ibrpp`
- `ibcac` and `ibgts`
- Any of the NI-488.2 routines that issue GPIB command bytes: `SendCmds`, `PPoll`, `Send`, and `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, ENOL 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 where 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.
- `ibeos` called with meaningless bits set in the high byte of the second parameter.
- `ibpad` or `ibsad` called with invalid addresses.
- `ibppc` called with invalid parallel poll configurations.
- A board-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 the GPIB configuration utility 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 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, or there is a system conflict with the base I/O address.

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.

EDMA (8)

EDMA occurs when an error occurs using DMA for data transfers. If your computer has more than 16MB of RAM, you are using DMA, the buffer is above 16 MB, and some error occurs when the driver attempts to remap the buffer to below 16 MB, the NI-488.2M software returns EDMA.

Solutions

- You can correct the EDMA problem in the software by using `ibdma` to disable DMA.
- You can correct the EDMA problem in the hardware by using the **Device Manager** to reconfigure the hardware to not use a DMA resource.

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. If any other call is attempted, EOIP is returned.

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 Windows 95 error code for this condition is contained in `ibcnt`.

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 GPIB configuration 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` and `FindRQS` 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.

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.

Appendix C
Customer Communication

Appendix
C

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.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call (512) 795-6990. You can access these services at:

United States: (512) 794-5422 or (800) 327-3077
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 1 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as anonymous and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the /`support` directories.

FaxBack Support

FaxBack is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access FaxBack from a touch-tone telephone at the following numbers:

(512) 418-1111 or (800) 329-7177

E-Mail Support (currently U.S. only)

You can submit technical support questions to the appropriate applications engineering team through e-mail at the Internet addresses listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

GPIB:	gpib.support@natinst.com
DAQ:	daq.support@natinst.com
VXI:	vxi.support@natinst.com
LabVIEW:	lv.support@natinst.com
LabWindows:	lw.support@natinst.com
HiQ:	hiq.support@natinst.com
VISA:	visa.support@natinst.com

Fax and Telephone Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

	Telephone	Fax
Australia	03 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

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___)_____ Phone (___)_____

Computer brand _____ Model _____ Processor _____

Operating system (include version number) _____

Clock Speed _____MHz RAM _____MB Display adapter _____

Mouse _____ yes _____ no Other adapters installed _____

Hard disk capacity _____MB Brand _____

Instruments used _____

National Instruments hardware product model _____ Revision _____

Configuration _____

National Instruments software product _____ Version _____

Configuration _____

The problem is _____

_____

_____

_____

List any error messages _____

_____

_____

_____

The following steps will reproduce the problem _____

_____

_____

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-488.2M™ User Manual for Windows 95

Edition Date: November 1995

Part Number: 321037A-01

Please comment on the completeness, clarity, and organization of the manual.

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-488.2M 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. 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	Controller-In-Charge. 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. The GPIB command used to reset the device or internal functions of all devices. See *SDC*.
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.
DLL	Dynamic link library.
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. It is the GPIB command used to trigger a device or internal function of an addressed Listener.
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 perhaps a 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.
GTL	Go To Local. It 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.

	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 I/O function, this global variable contains the actual number of bytes transmitted.
`iberr`	A global variable that contains the specific error code associated with a function call that failed.
`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.
Listener	A GPIB device that receives data messages from a Talker.
LLO	Local Lockout. The GPIB command used to tell all devices that they may or should ignore remote (GPIB) data messages or local (front panel) controls, depending on whether the device is in local or remote program mode.
low-level function	A rudimentary board or device function that performs a single operation.

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.
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. 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. It is the GPIB command used to configure an addressed Listener to participate in polls.
PPD	Parallel Poll Disable. It is the GPIB command used to disable a configured device from participating in polls. There are 16 PPD commands.
PPE	Parallel Poll Enable. It 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. It 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-488.2M software and the user application must resynchronize after asynchronous I/O operations have completed.
RQS	Request Service.

S

s	Seconds.
SDC	Selected Device Clear. The GPIB command used to reset internal or device functions of an addressed Listener. See *DCL*.
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	Serial Poll Disable. The GPIB command used to cancel an SPE command.
SPE	Serial Poll Enable. The GPIB command used to enable a specific device to be polled. That device must also be addressed to talk. See *SPD*.
SRQ	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-488.2M 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. The GPIB command used to pass control of the bus from the current Controller to an addressed Talker.
timeout	A feature of the NI-488.2M 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.
UNL	Unlisten. The GPIB command used to unaddress any active Listeners.
UNT	Untalk. The GPIB command used to unaddress an active Talker.

Index

Numbers/Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V

W