GPIB

NI-488.2M™ User Manual for Windows 95 and Windows NT

June 1996 Edition
Part Number 321037B-01

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`

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

Bulletin Board Support

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

FaxBack Support

(512) 418-1111

Telephone Support (U.S.)

Tel: (512) 795-8248
Fax: (512) 794-5678

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 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico 95 800 010 0793, Netherlands 0348 433466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200, U.K. 01635 523545

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

Trademarks

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

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

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
GPIB Spy Utility

Chapter 6
Win32 Interactive Control Utility

Chapter 7
GPIB Programming Techniques

Chapter 8
GPIB Configuration Utility

Appendix A
Status Word Conditions

Appendix B
Error Codes and Solutions

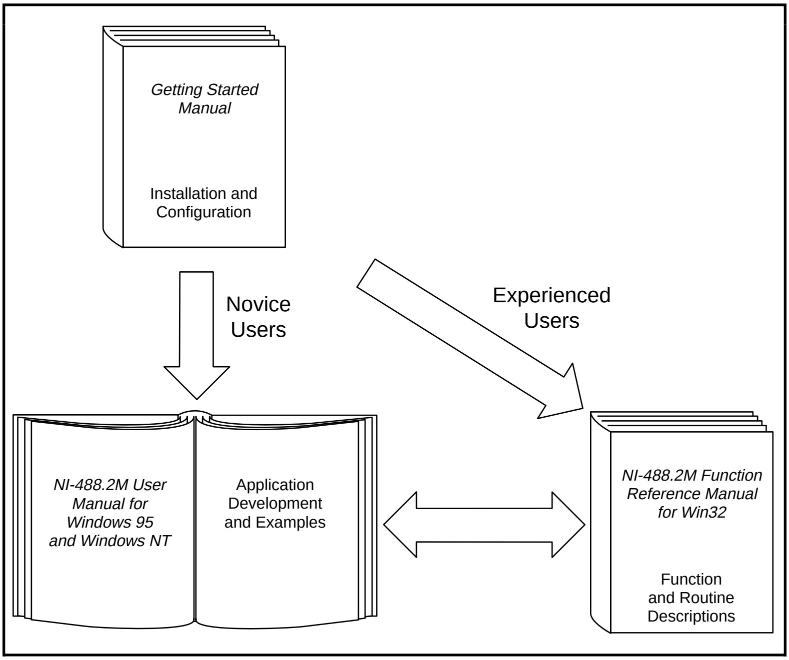
Figures

Tables

About This Manual

This manual describes the features and functions of the NI-488.2M software for both Windows 95 and Windows NT. The NI-488.2M software for Windows 95 is meant to be used with the Microsoft Windows 95 operating system. The NI-488.2M software for Windows NT is meant to be used with the Microsoft Windows NT (version 3.51 and later) operating system. This manual assumes that you are already familiar with the appropriate Microsoft operating system.

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 and Windows NT* 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.
- Chapter 3, *Developing Your Application*, explains how to develop a GPIB application using NI-488 functions and NI-488.2 routines.
- Chapter 4, *Debugging Your Application*, describes several ways to debug your application.
- Chapter 5, *GPIB Spy Utility*, introduces you to GPIB Spy, the application monitor you can use to monitor NI-488 and NI-488.2 calls.
- Chapter 6, *Win32 Interactive Control Utility*, introduces you to Win32 Interactive Control, the interactive control utility that you can use to communicate with GPIB devices interactively.
- Chapter 7, *GPIB Programming Techniques*, describes techniques for using some NI-488 functions and NI-488.2 routines in your application.
- Chapter 8, *GPIB Configuration Utility*, contains a description of the GPIB configuration utility you can use to configure your 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, *Windows 95: Troubleshooting and Common Questions*, describes how to troubleshoot problems and lists some common questions for Windows 95 users.
- Appendix D, *Windows NT: Troubleshooting and Common Questions*, describes how to troubleshoot problems and lists some common questions for Windows NT users.
- Appendix E, *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.
bold italic	Bold italic text denotes a note, caution, or warning.
`bold monospace`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen.
italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept.
`italic monospace`	Italic text in this font denotes that you must supply the appropriate words or values in the place of these items.
`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.
< >	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 488.2-1992, 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 Windows NT 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 E, *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 488.2-1992 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.

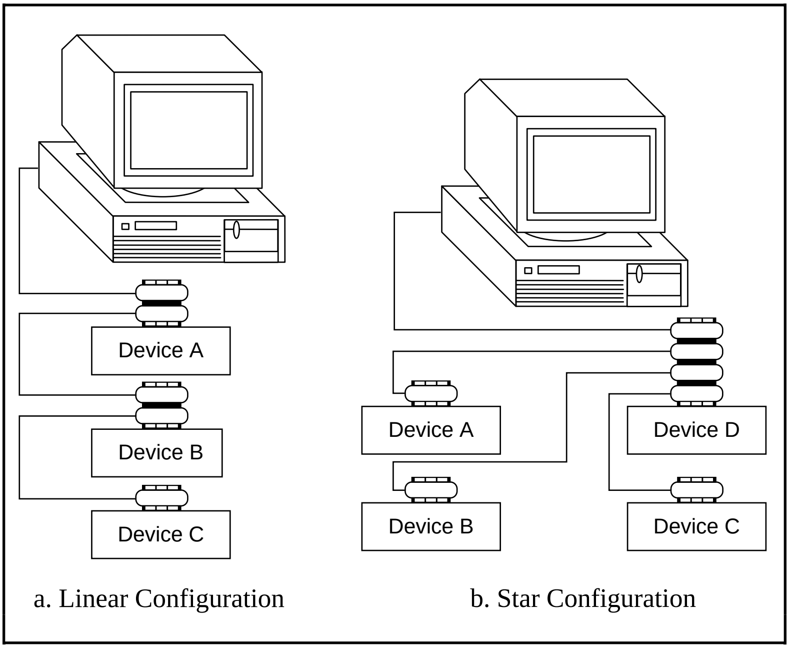


Figure 1-2. Linear and Star System Configuration

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.

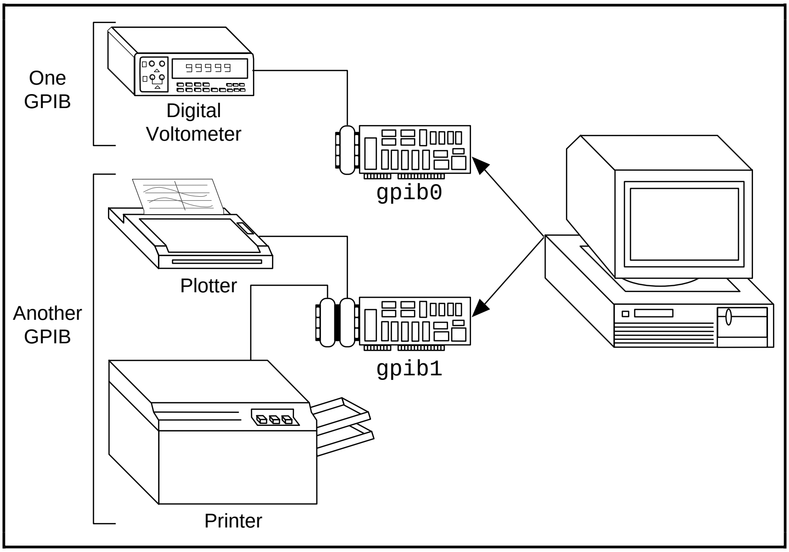


Figure 1-3. Example of Multiboard System Setup

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 for Windows 95

NI-488.2M Software for Windows 95 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:

- A documentation file, `readme.txt`, 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: 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.
- A Win32 dynamic link library, `gpib-32.dll`, 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.

- GPIB Spy is the GPIB application monitor program. It is a debugging tool that you can use to monitor the NI-488.2 calls your GPIB applications make.
- 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.
- Diagnostic is a utility that you use to verify that the GPIB hardware and software have been installed properly.

16-Bit Windows Support Files

- A 16-bit Windows dynamic link library, `gpib.dll`, used 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.
- A 32-bit Windows dynamic link library, `gpib32ft.dll`, 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`.

DOS Support Files

- A Virtual Device Driver (VxD), `gpibdosk.vxd`, that serves as the DOS device driver, to trap NI-488 function calls and NI-488.2 routine calls made by DOS applications and route them to the standard 32-bit dynamic link library, `gpib-32.dll`. This file replaces the real-mode DOS device driver that would be loaded from your `config.sys` file if you were using the DOS environment for DOS GPIB applications.
- A Win32 executable, `gpibdos.exe`, that helps `gpibdosk.vxd` thunk DOS GPIB calls to 32-bit calls that address the standard 32-bit dynamic link library, `gpib-32.dll`.

Microsoft C/C++ Language Interface Files

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

Borland C/C++ Language Interface Files

- A documentation file, `readme.txt`, that contains information about the C language interface.
- A 32-bit include file, `decl-32.h`, that contains NI-488 function and NI-488.2 routine prototypes and various predefined constants.
- A 32-bit C language interface file, `borlandc_gpib-32.obj`, that an application links with in order to access the 32-bit DLL.

Microsoft Visual Basic Language Interface Files

- A documentation file, `readme.txt`, that contains information about the Visual Basic language interface.
- A Visual Basic global module, `niglobal.bas`, that contains certain predefined constant declarations.
- A Visual Basic source file, `vbib-32.bas`, that contains 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 or directly access the functions exported by the DLL.
- A Win16 application uses the 16-bit thunking DLL (`gpib.dll`) and 32-bit thunking DLL (`gpib32ft.dll`) to access the GPIB driver.
- A DOS application uses the DOS support VxD and application to access the GPIB driver.

Figure 1-4 shows the interaction between various types of GPIB applications (shaded sections) and the NI-488.2M software components.

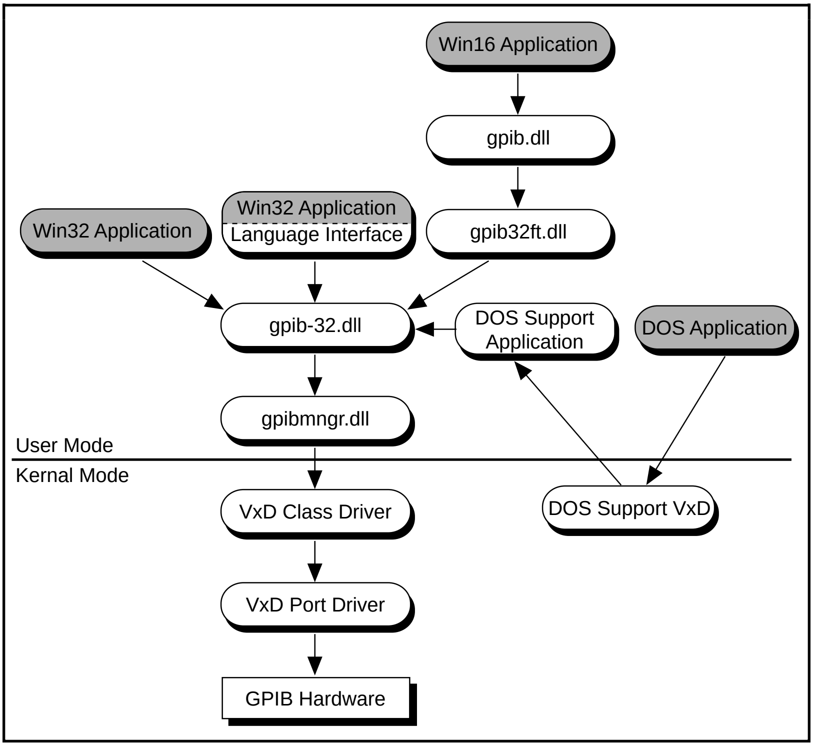


Figure 1-4. How the NI-488.2M Software Works with Windows 95

Uninstalling the GPIB Hardware from Windows 95

Before physically removing the 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.

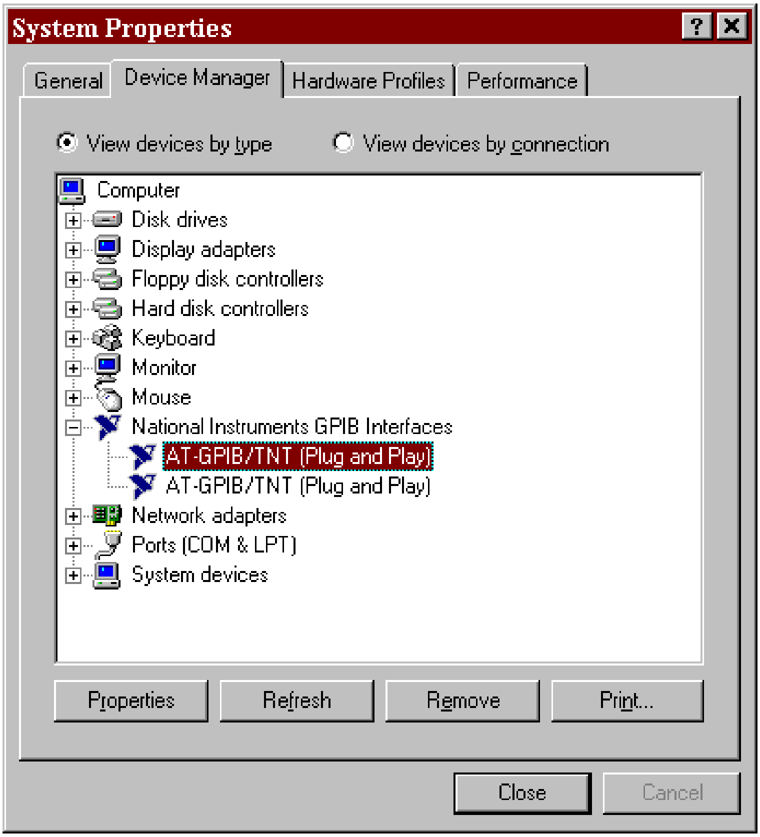


Figure 1-5. Selecting an Interface to Remove from Windows 95

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

Uninstalling the GPIB Software for Windows 95

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 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.

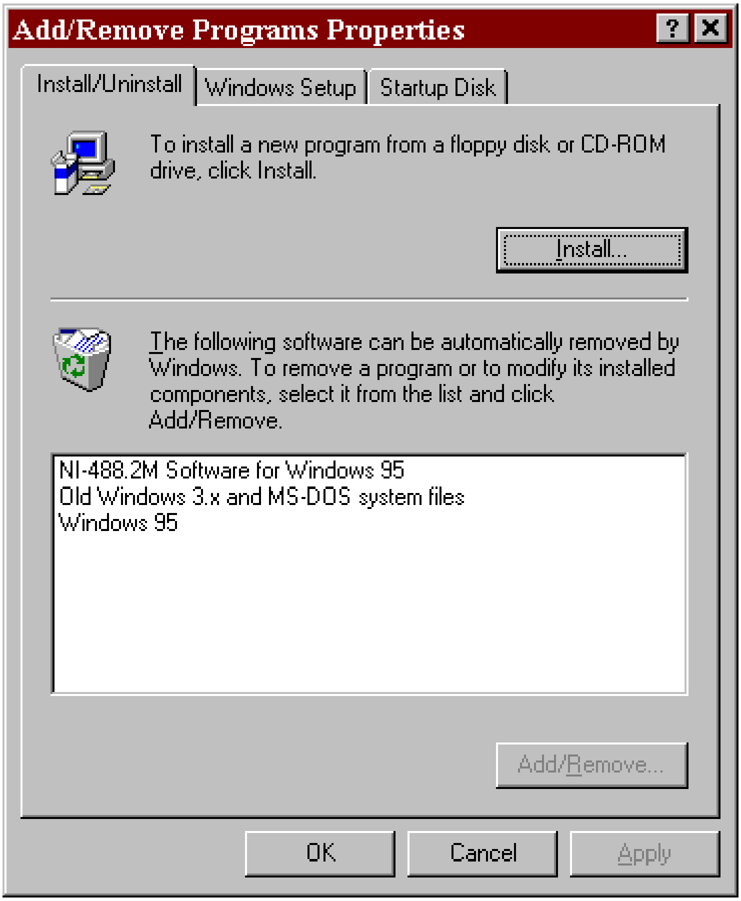


Figure 1-6. Add/Remove Programs Properties Dialog Box in Windows 95

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.

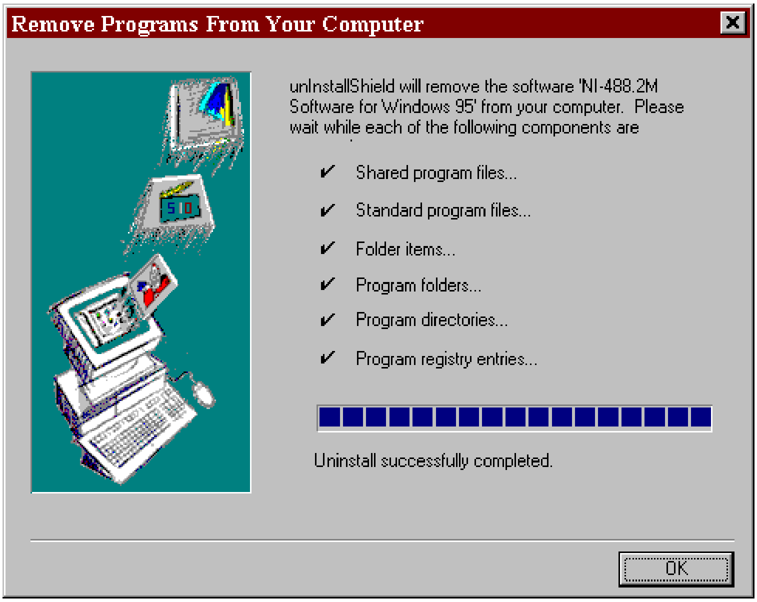


Figure 1-7. Uninstallation Results in Windows 95

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.

The NI-488.2M Software for Windows NT

NI-488.2M Software for Windows NT Components

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

NI-488.2M Driver and Driver Utilities

The distribution disk contains the following driver and utility files:

- A documentation file, `readme.txt`, 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 Windows NT kernel driver components.
- A Win32 dynamic link library, `gpib-32.dll`, that acts as the interface between all applications and the kernel mode GPIB driver.
- Win32 Interactive Control 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.
- GPIB Spy is the GPIB application monitor program. It is a debugging tool that you can use to monitor the NI-488.2 calls your GPIB applications make.
- The GPIB configuration utility, a control panel application that you use to modify the software configuration parameters of the NI-488.2M software.
- Diagnostic is a utility you can use to verify that the GPIB hardware and software have been installed properly.

DOS and 16-Bit Windows Support Files

- A documentation file, `readme.txt`, that contains information about using existing DOS and 16-bit Windows applications under Windows NT.
- A Virtual device driver, `gpib-vdd.dll`, that allows existing NI-488.2 for DOS and 16-bit Windows applications to access the NI-488.2M software.

- A DOS device driver, `gpib-nt.com`. When you run an existing NI-488.2 application for DOS in the Windows NT environment, this file replaces the `gpib.com` driver that you used in the DOS environment.
- A Windows dynamic link library, `gpib.dll`. When you run an existing NI-488.2 application for Windows in the Windows NT environment, this file replaces the GPIB DLL that you used in the Windows (16-bit) environment.

Microsoft C/C++ Language Interface Files

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

Borland C/C++ Language Interface Files

- A documentation file, `readme.txt`, that contains information about the C language interface.
- A 32-bit include file, `decl-32.h`, that contains NI-488 function and NI-488.2 routine prototypes and various predefined constants.
- A 32-bit C language interface file, `borlandc_gpib-32.obj`, that an application links with in order to access the 32-bit DLL.

Microsoft Visual Basic Language Interface Files

- A documentation file, `readme.txt`, that contains information about the Visual Basic language interface.
- A Visual Basic global module, `niglobal.bas`, that contains certain predefined constant declarations.
- A Visual Basic source file, `vbib-32.bas`, that contains 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 NT

The main components of the NI-488.2M software are a dynamic link library that runs in user mode and a layered NT device driver that runs in kernel mode. The layered NT device driver consists of three drivers: a device class driver that handles device-level calls, a board class driver that handles board-level calls, and a GPIB port driver that uses the Hardware Abstraction Layer (HAL) to communicate with the GPIB hardware. The top two layers of the layered NT device driver are accessed from user mode by `gpib—32.dll`, a 32-bit Windows NT 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 or directly access the functions exported by the DLL.
- A Win16 application uses the 16-bit DLL (`gpib.dll`) to access the GPIB virtual device driver (`gpib-vdd.dll`).
- A DOS application uses the DOS device driver (`gpib—nt.com`) to access the GPIB virtual device driver.

Figure 1-8 shows the interaction between various types of GPIB applications and the NI-488.2M software components.

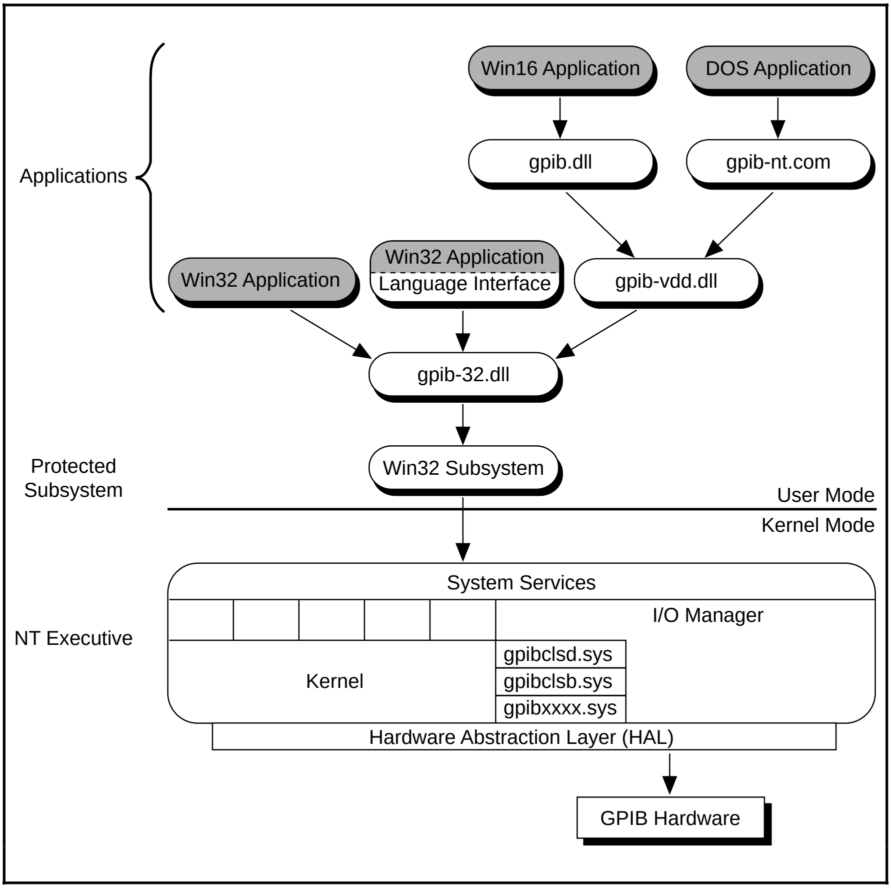


Figure 1-8. How the NI-488.2M Software Works with Windows NT

Unloading and Reloading the NI-488.2M Driver for Windows NT

You can unload and restart the NI-488.2M driver using the GPIB configuration utility.

To run this utility in Windows NT 3.51, double-click on the **GPIB** icon in the **Control Panel**, which is located in the **Main** group of the **Program Manager**. To run this utility in Windows NT 4.0 or later, select **Start»Settings»Control Panel**, and double-click on the **GPIB** icon.

The main window has an **Unload** button and a **Restart** button. If you click on the **Unload** button, the NI-488.2M driver is unloaded. If you click on the **Restart** button, the NI-488.2M driver is automatically unloaded and then reloaded. Refer to Chapter 8, *GPIB Configuration Utility*, for a more complete description.

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 and Visual Basic source code for each of the nine examples. The programs are listed in order of increasing complexity. If you are new to GPIB programming, you might want to study the contents and concepts of the first sample, `simple.c`, before moving on to more complex examples.

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

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

- `rqs4882.c` is the source code file for Example 7. It uses NI-488.2 routines to communicate with GPIB devices that use the GPIB SRQ line to request service.
- `ppoll.c` is the source code file for Example 8. It uses NI-488.2 routines to conduct parallel polls.
- `non_cic.c` is the source code file for Example 9. It illustrates how you can use the NI-488.2M 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.

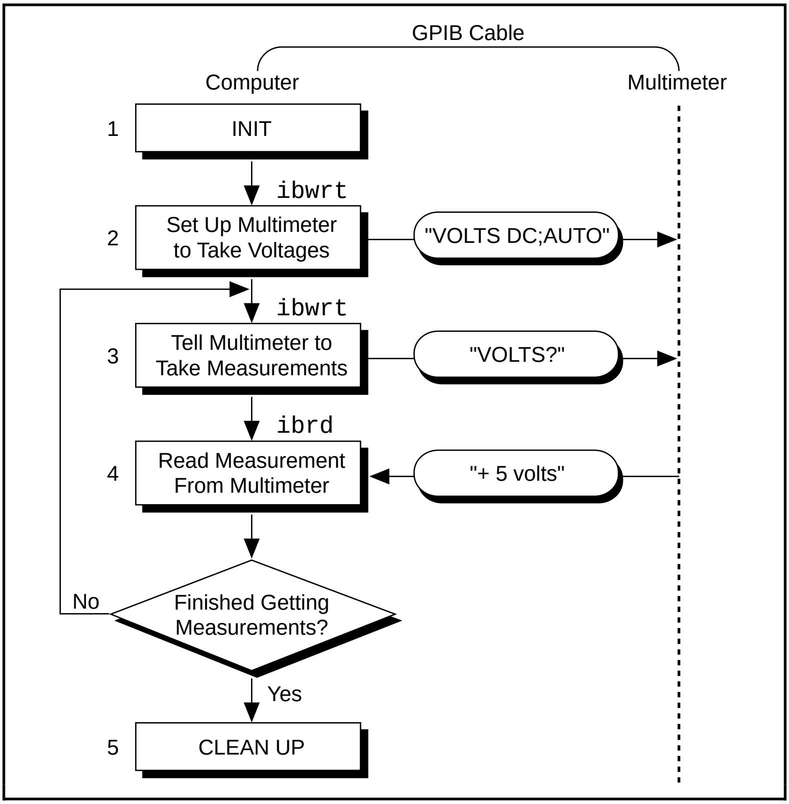


Figure 2-1. Program Flowchart for Example 1

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.

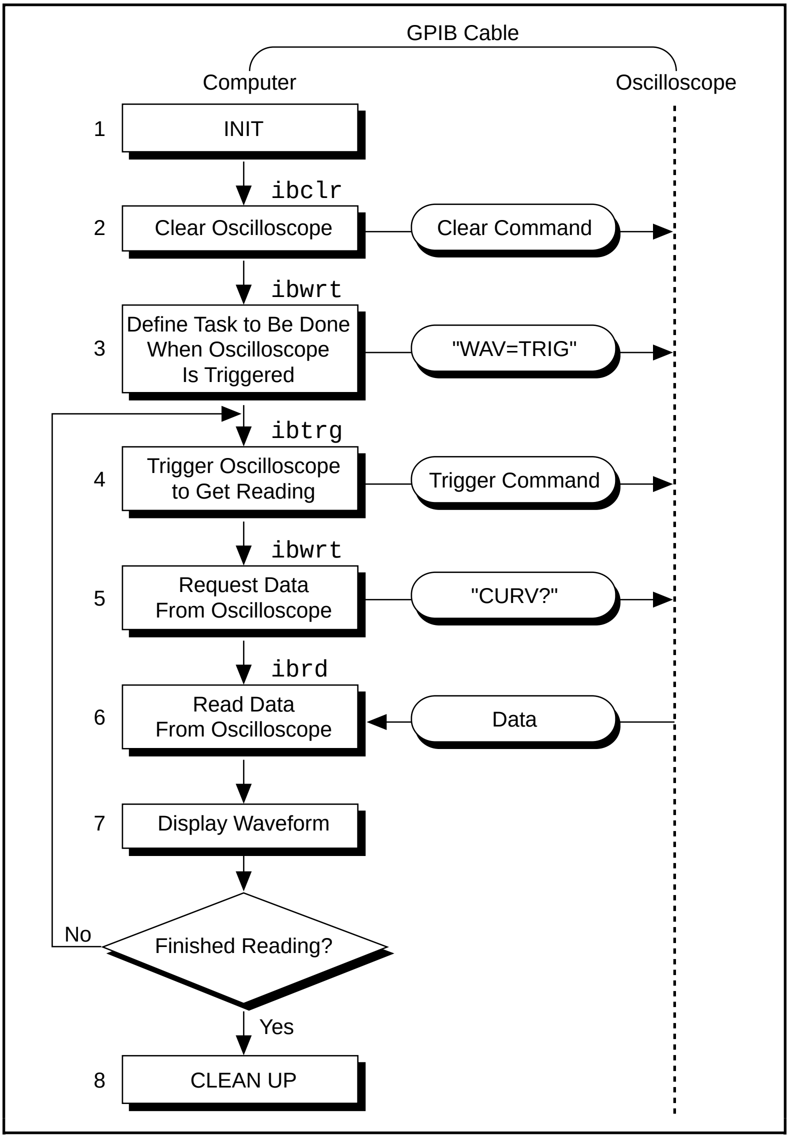


Figure 2-2. Program Flowchart for Example 2

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 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.

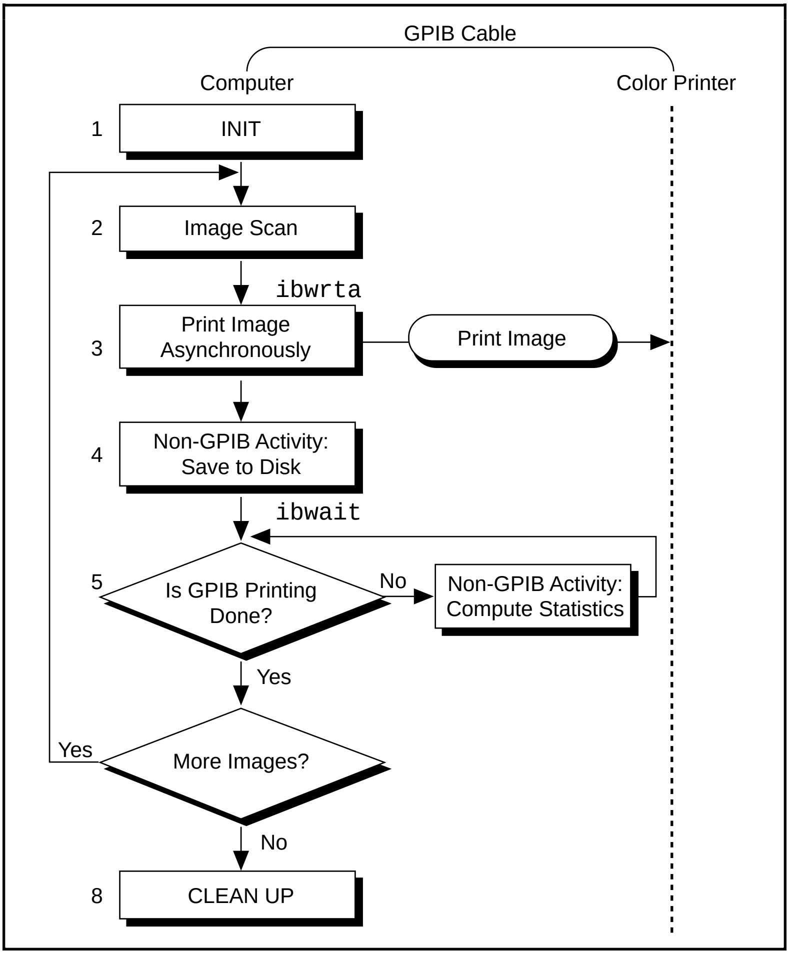


Figure 2-3. Program Flowchart for Example 3

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.

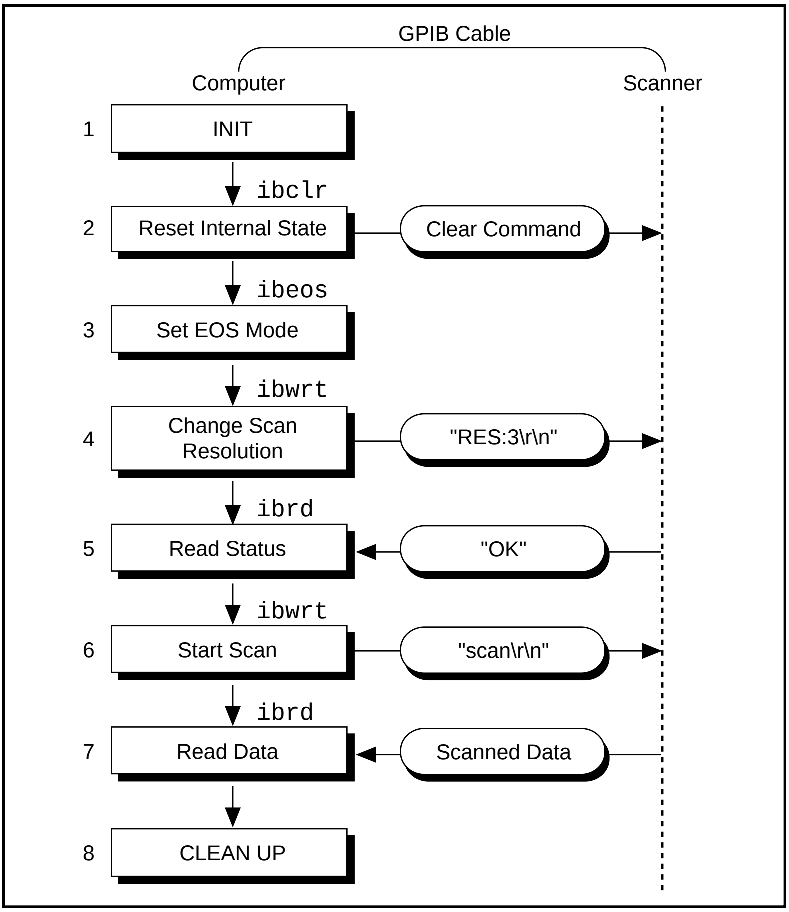


Figure 2-4. Program Flowchart for Example 4

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.

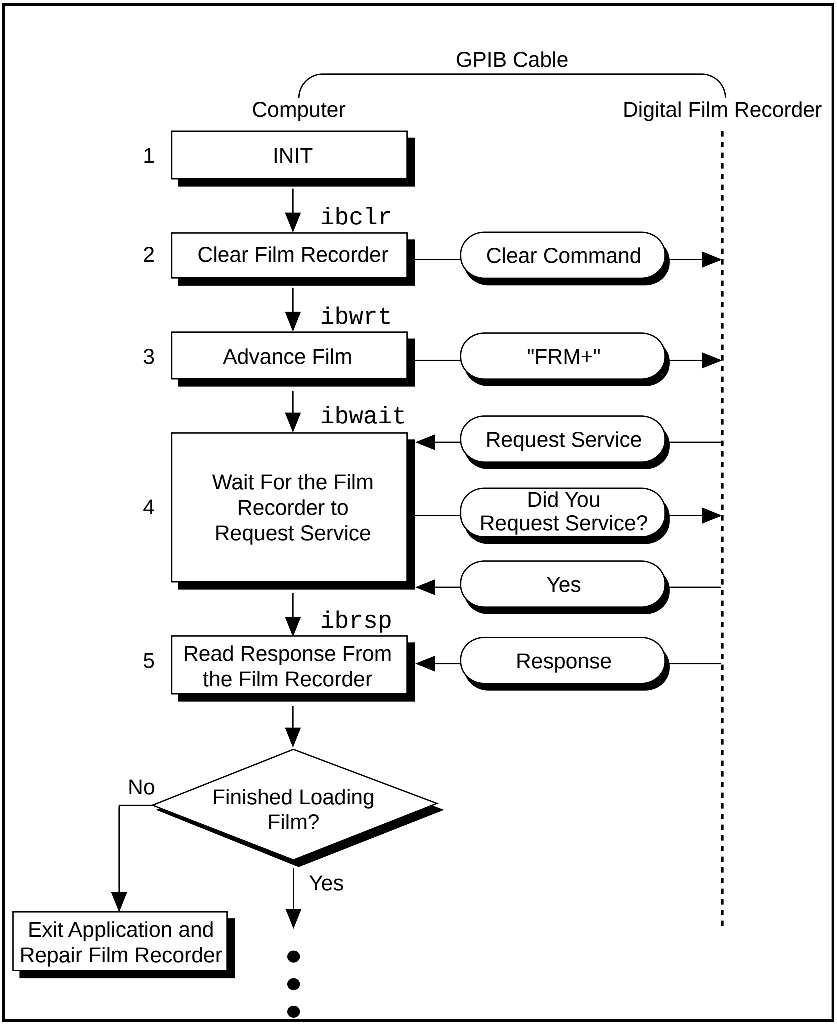
6. A color image transfers to the digital film recorder in three consecutive passes—one pass each for the red, green and blue components of the image. Sub-steps 6a, 6b, and 6c are repeated for each of the passes:

   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)

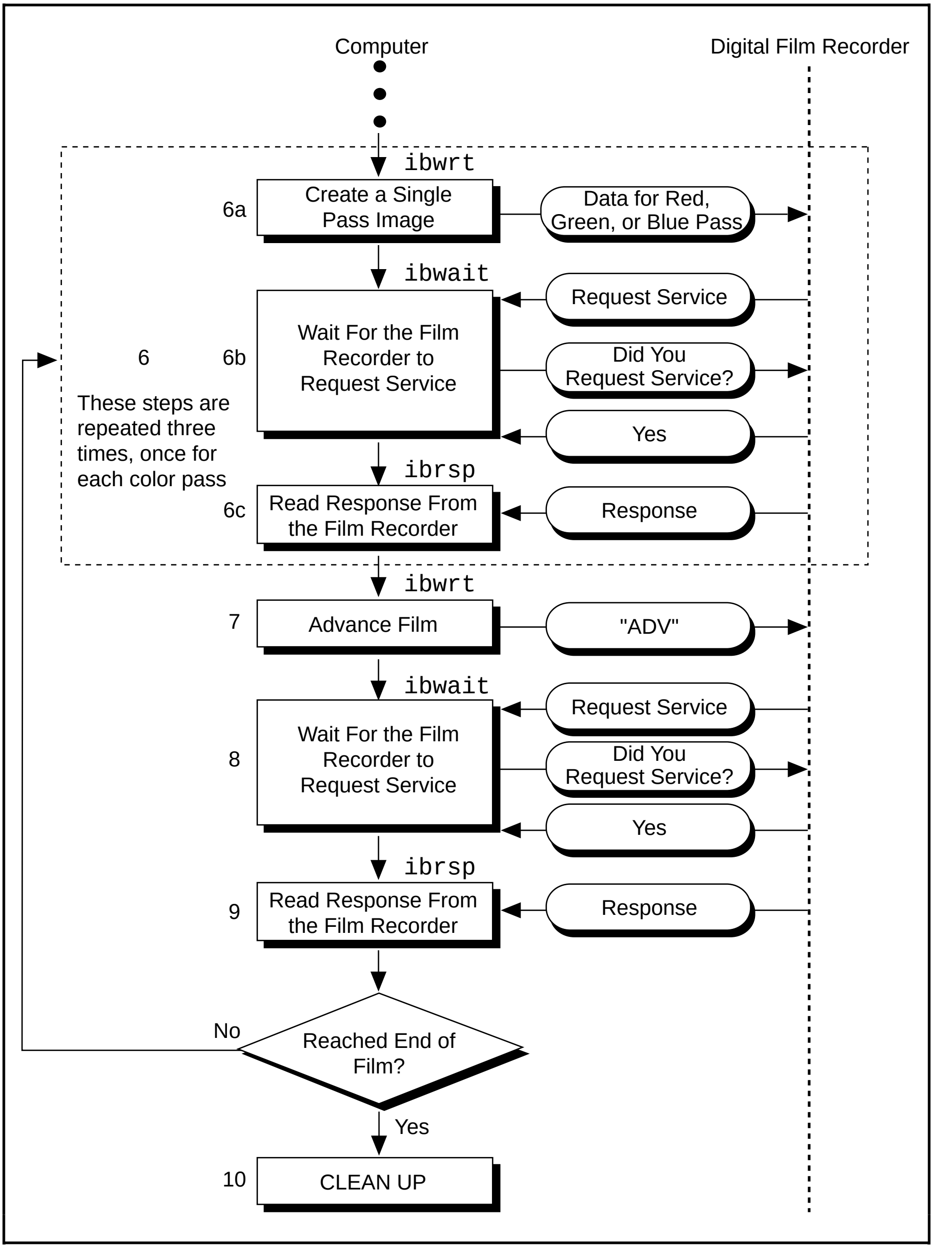


Figure 2-5. Program Flowchart for Example 5 (Continued)

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.

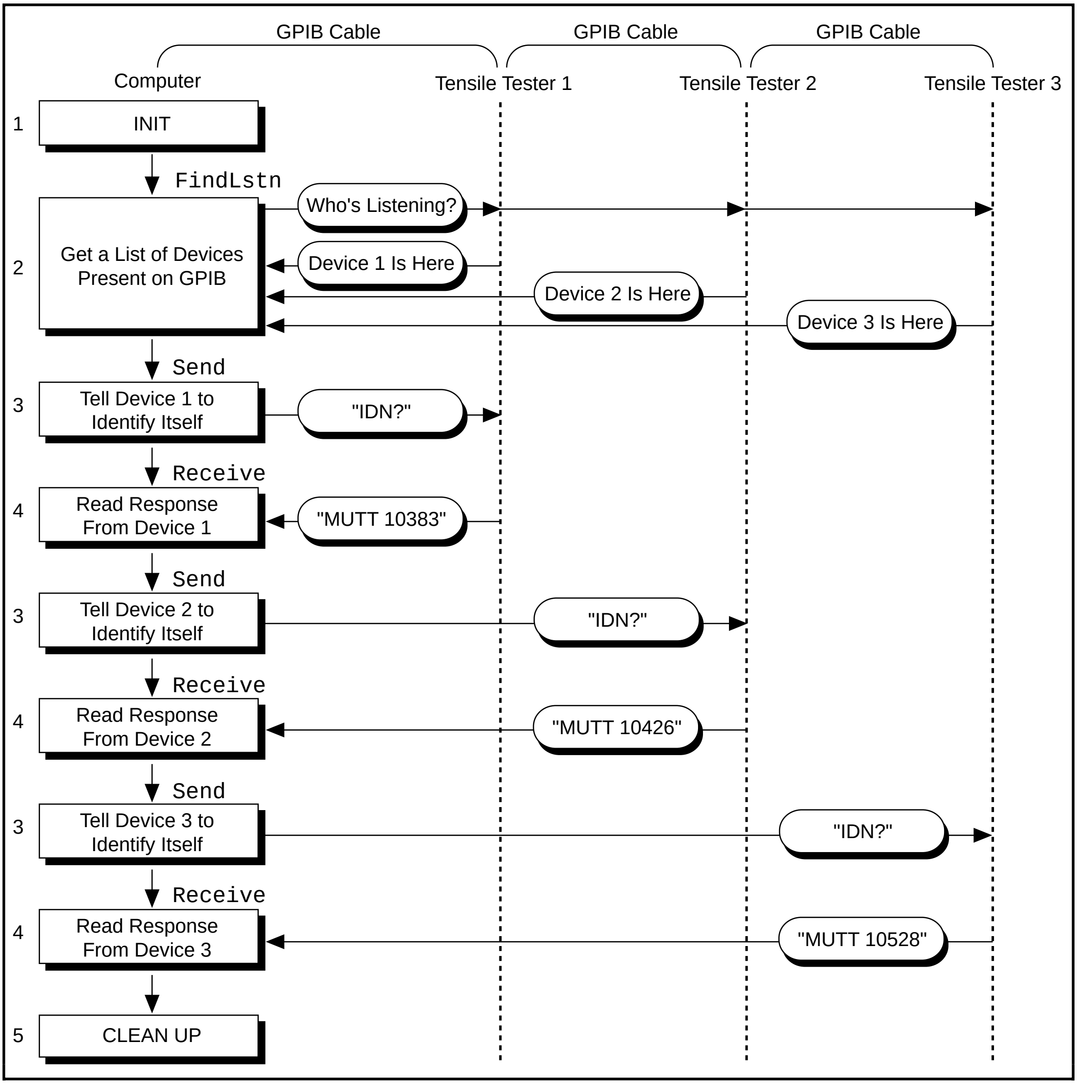


Figure 2-6. Program Flowchart for Example 6

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.

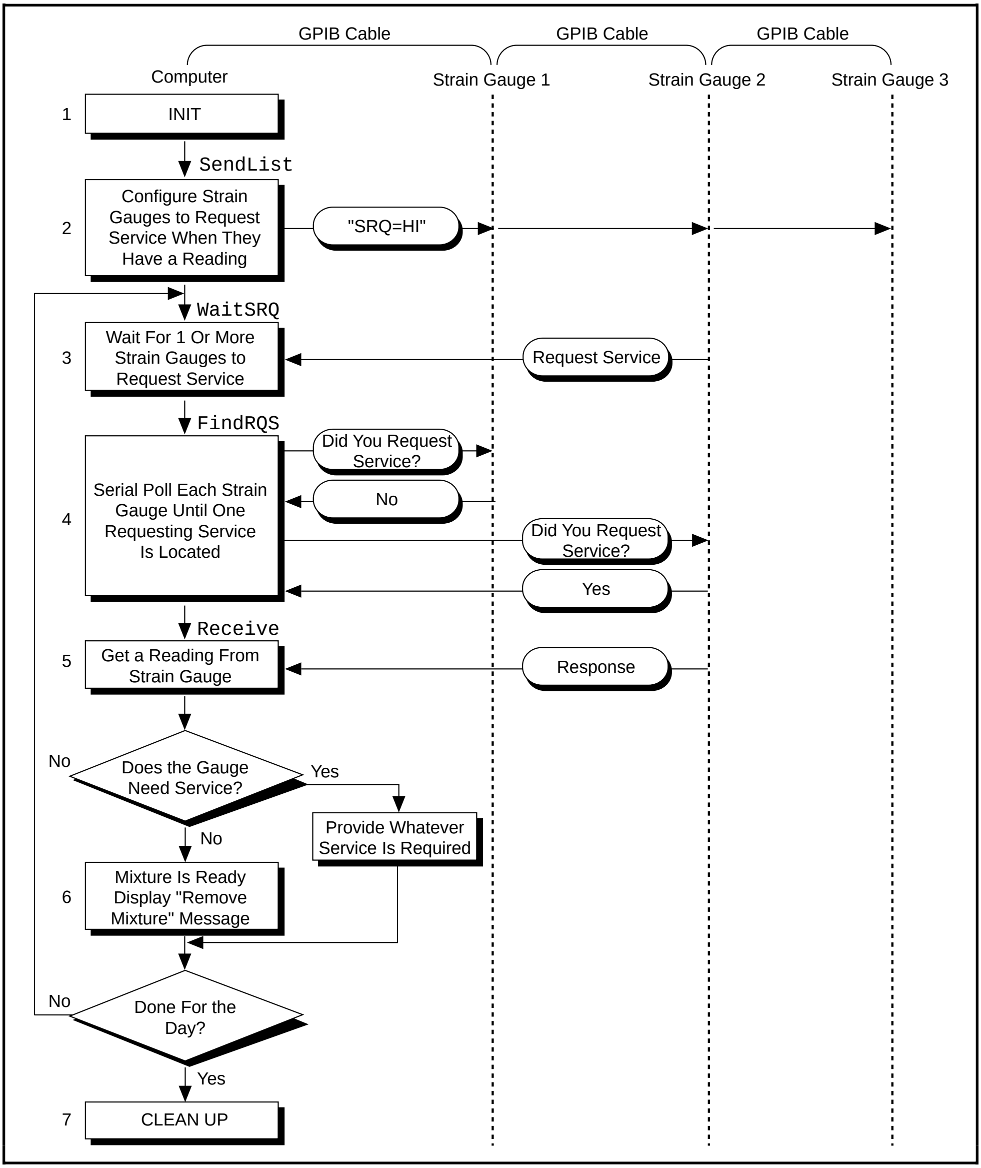


Figure 2-7. Program Flowchart for Example 7

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.

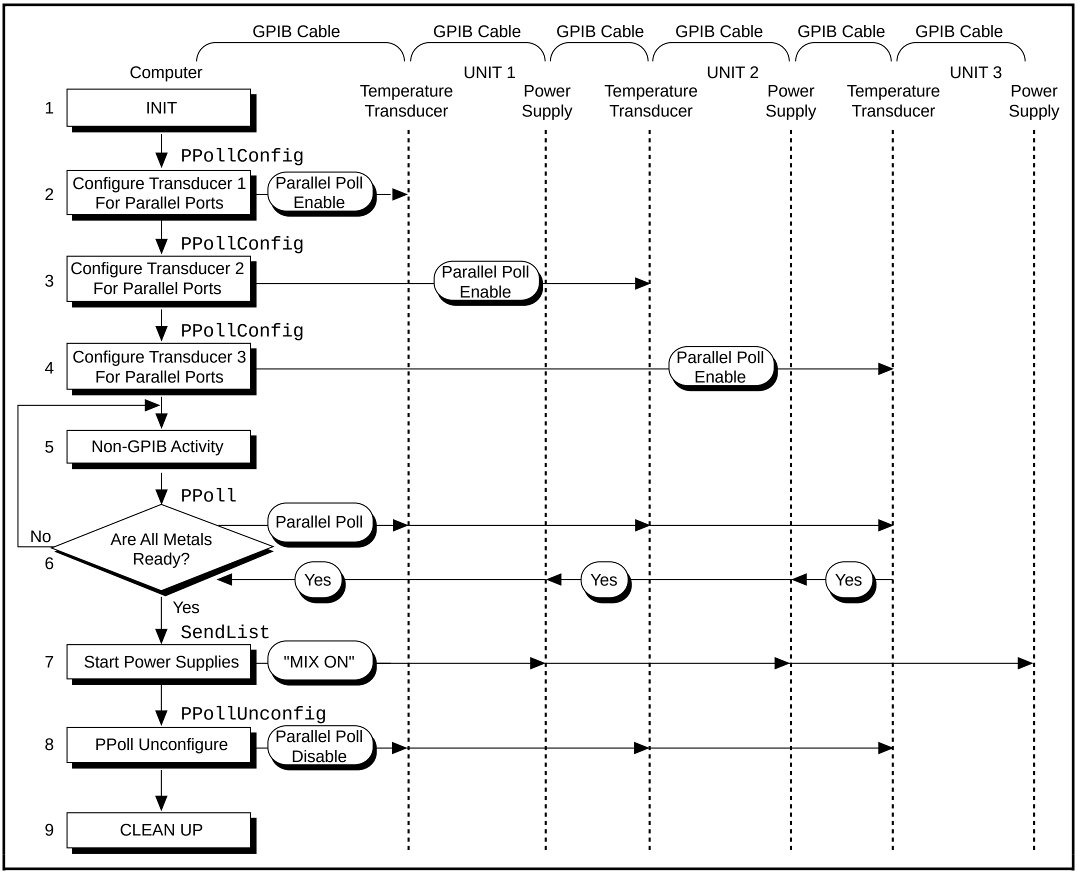


Figure 2-8. Program Flowchart for Example 8

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.

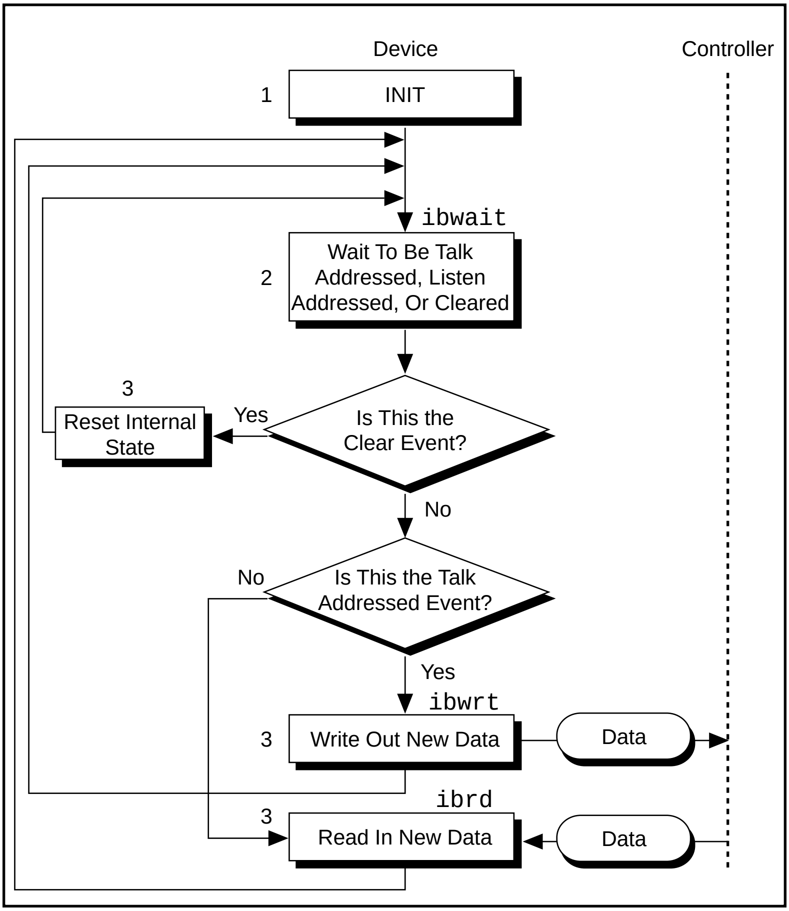


Figure 2-9. Program Flowchart for Example 9

Chapter 3

Developing Your Application

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

Choosing Your Programming Methodology

Based on your development environment, you can select a method for accessing the driver, and based on your GPIB programming needs, you can choose between the NI-488 functions and NI-488.2 routines.

Choosing a Method to Access the NI-488.2M Driver

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

NI-488.2M Language Interfaces

You can use a language interface if your program is written in Microsoft Visual C/C++ (2.0 or higher), Borland C/C++ (4.0 or higher), or Microsoft Visual Basic (4.0 or higher). Otherwise, you must access the `gpib-32.dll` directly.

Direct Entry Access

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 and all of the NI-488 and NI-488.2 calls.

Choosing between NI-488 Functions and NI-488.2 Routines

The NI-488.2M software includes 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 applications, 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 devices 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-Level Functions

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

NI-488 Board-Level Functions

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

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

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

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

When your system includes a board that must access 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

You can mix board-level NI-488 functions with the NI-488.2 routines to have access to all of the NI-488.2 functionality.

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. These global status variables are the status word (`ibsta`), the error variable (`iberr`) and the count variables (`ibcnt` and `ibcntl`). They contain useful information about the performance of your application. Your application should check these variables after each GPIB call. The following sections describe each of these global variables and how you can use them in your application.

☞ **Note:** ***If your application is a multithreaded application, refer to the*** *Writing Multithreaded Win32 GPIB Applications* ***section in Chapter 7,*** *GPIB Programming Techniques.*

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 examine various status bits in `ibsta` and use that information to make decisions about continued processing. If you check for possible errors after each call using the `ibsta` ERR bit, 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 defines each of the `ibsta` status bits. You can test for an `ibsta` status bit being set using the bitwise `and` operator (“&” in C/C++). For example, the `ibsta` ERR bit is bit 15 of `ibsta`. To check for a GPIB error, use the following statement after each GPIB call as shown:

```
if (ibsta & ERR)
   printf("GPIB error encountered");
```

Error Variable – iberr

If the ERR bit is set in `ibsta`, a GPIB error has occurred. When an error occurs, the error type is specified by `iberr`. To check for a GPIB error, use the following statement after each GPIB call:

```
if (ibsta &ERR)
   printf("GPIB error %d encountered", 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. In Win32 applications, `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 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();
}
```

Using Win32 Interactive Control to Communicate with Devices

Before you begin writing your application, 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. 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 6, *Win32 Interactive Control Utility*.

Programming Model for NI-488 Applications

This section discusses items you should include in your application, general program steps, and an NI-488 example.

Items to Include

- In a C application, include the header files `windows.h` and `decl-32.h`. `windows.h`, the standard Windows header file, contains definitions used by `decl-32.h` and `decl-32.h` contains prototypes for the GPIB functions and constants that you can use in your application.
- 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 using NI-488 functions. The flowchart is for device-level calls.

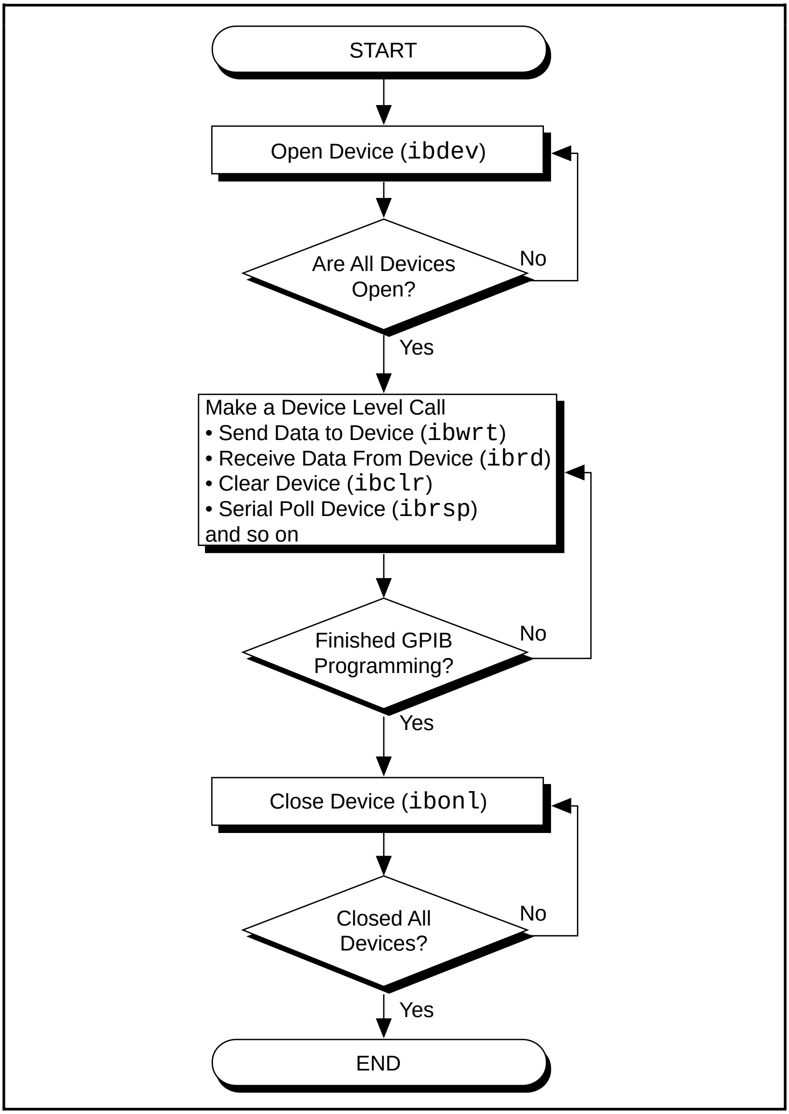


Figure 3-1. General Program Shell Using NI-488 Device Functions

NI-488 General Program Steps and Examples

The following steps demonstrate how to use the NI-488 device functions in your application. The NI-488.2M software includes the source code for an example written in C (`devsamp.c`) and the source code for the example written to use direct entry to access the `gpib-32.dll` (`dlldev.c`). The NI-488.2M software also includes a sample program written in Visual Basic, `devsamp.frm`.

Step 1. Open a Device

Your first NI-488 function call should be a call to `ibdev` to open a device. The `ibdev` function requires the following parameters:

- Connect board index (typically set to 0, because your board is GPIB0),
- Primary address for the GPIB instrument (refer to the GPIB instrument manual)
- Secondary address for the GPIB instrument (0 if the GPIB instrument does not use secondary addressing)
- Timeout period (typically set to T10s which is 10 seconds)
- End-of-transfer mode (typically 1 so that EOI is asserted with the last byte of writes)
- EOS detection mode (0 if the GPIB instrument does not use EOS characters)

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. A successful `ibdev` call returns a unit descriptor handle, `ud`, that is used for all NI-488 calls that communicate with the GPIB instrument.

Step 2. Clear the Device

Use `ibclr` to clear the device before you configure the device for your application. Clearing the device resets its internal functions to a default state.

Step 3. Communicate with the Device

After you open and clear the device, your GPIB instrument is ready to receive instructions. If you want to acquire readings from your device, you can do so in several ways . Each GPIB device has its own specific instructions. You should refer to the documentation that came with your GPIB device to learn how to properly communicate with it. For this example, assume that the GPIB device can be programmed to acquire readings whenever it is triggered. Furthermore, assume that the GPIB device requests service when it has acquired a reading. Given these assumptions, the following steps are necessary:

Step 3a.

Program the GPIB device to acquire a reading whenever it receives a GPIB trigger using the `ibwrt` function. The buffer that you pass to `ibwrt` is the command message that programs the device to behave properly.

Step 3b.

Trigger the device using the `ibtrg` function.

Step 3c.

Wait for the device to acquire the reading using the `ibwait` function with a `mask` value of `RQS | TIMO` because the event of interest is the device's ReQuest for Service (RQS). If the `ibwait` function times out before the RQS event occurs, the timeout bit (TIMO) is set in the `ibsta` value for the call.

Step 3d.

If the wait for the service request succeeded, get the device's serial poll response byte and verify that it indicates that the device obtained a good measurement, using the `ibrsp` function.

Step 3e.

Read the measurement from the device using the `ibrd` function and record it in a list of device measurements.

Steps 3b through 3e should be repeated for each measurement you want to acquire.

Step 4. Place the Device Offline Before Exiting Your Application

Once you are finished accessing the GPIB device, take it offline using the `ibonl` function before you exit your application.

Programming Model for NI-488.2 Applications

This section discusses items you should include in an application that uses NI-488.2 routines, general program steps, and an NI-488.2 example.

Items to Include

- In a C application, include the header files `windows.h` and `decl-32.h`. `windows.h`, the standard Windows header file, contains definitions used by `decl-32.h` and `decl-32.h` contains prototypes for the GPIB routines and constants that you can use in your application.
- 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 using NI-488.2 routines.

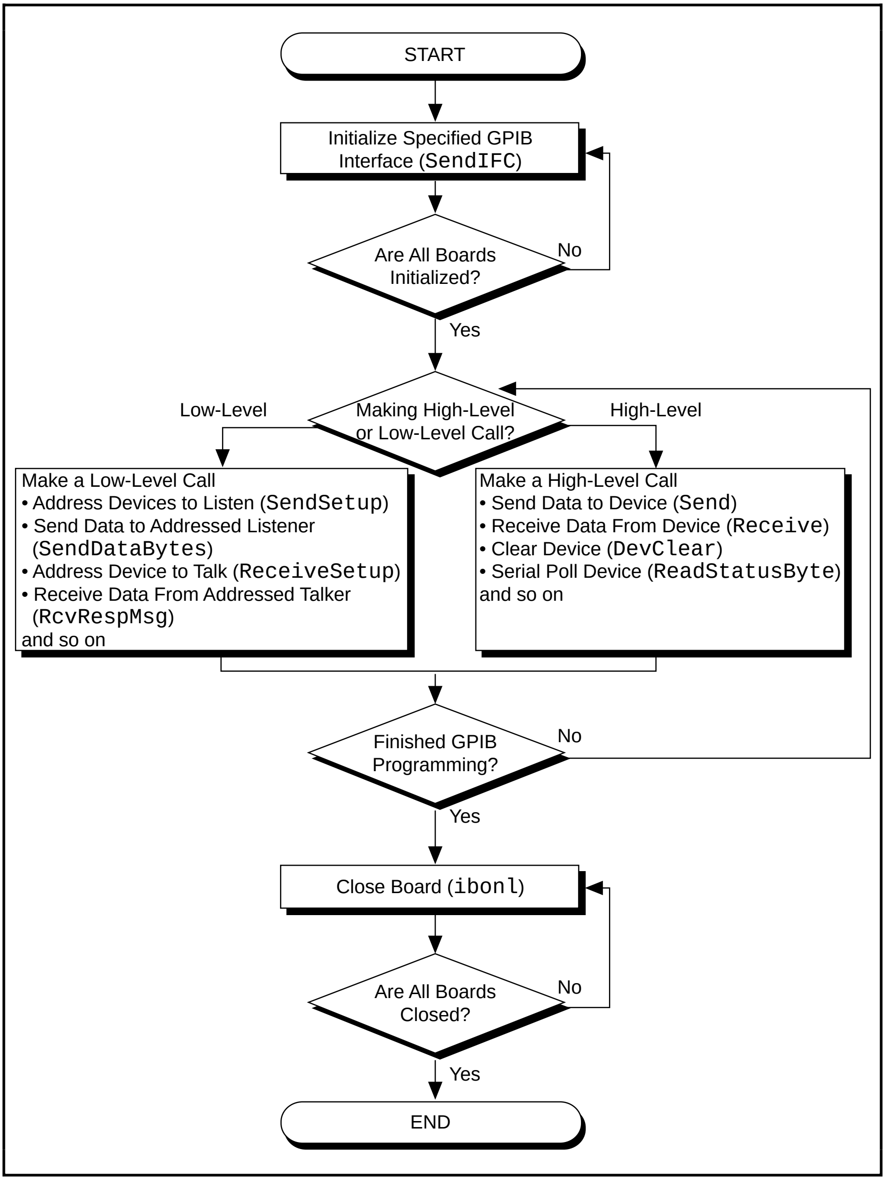


Figure 3-2. General Program Shell Using NI-488.2 Routines

NI-488.2 General Program Steps and Examples

The following steps demonstrate how to use the NI-488.2 routines in your application. The NI-488.2M software includes the source code for an example written in C (`samp4882.c`) and the source code for the example written to use direct entry to access the `gpib-32.dll` (`dll4882.c`). The NI-488.2M software also includes a sample program written in Visual Basic, `samp4882.frm`.

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, typically 0 for GPIB0.

Step 2. Determine the GPIB Address of Your Device

If you do not know the address of your device, you can use the `FindLstn` routine to find all the devices attached to the GPIB. The `FindLstn` routine requires the following parameters:

- Interface board number (typically set to 0, because your board is GPIB0)
- A list of primary addresses, terminated with the NOADDR constant
- A list of GPIB addresses of devices found listening on the GPIB
- Limit which is the number of the GPIB addresses to report

The `FindLstn` routine tests for the presence of all of the primary addresses that are passed to it. If a device is present at a particular primary address, then the primary address is stored in the GPIB addresses list. Otherwise, all secondary addresses of the given primary address are tested, and the GPIB address of any devices found are stored in the GPIB addresses list. Once you have the list of GPIB addresses, you can determine which one corresponds to your instrument and use it for subsequent NI-488.2 calls.

Alternatively, if you already know your GPIB device's primary and secondary address, you can create an appropriate GPIB address to use in subsequent NI-488.2 calls as follows: a GPIB address is a 16-bit value that contains the primary address in the low byte and the secondary address in the high byte. If you are not using secondary addressing, the secondary address is 0. For example, if the primary address is 1, then the word = 0x01; otherwise if the primary address is 1 and the secondary address is 0x67, then the word = 0x6701.

Step 3. Initialize the Device

After you find the device, use the `DevClear` routine to clear it. The first argument is the GPIB board number. The second argument is the GPIB address as determined in Step 2.

Step 4. Communicate with the Device

After initialization, your GPIB instrument is ready to receive instructions. If you want to acquire readings from your device, you do so in several ways. Each GPIB device has its own specific instructions. You should refer to the documentation that came with your GPIB device to learn how to properly communicate with it. For this example, assume that the GPIB device can be programmed to acquire readings whenever it is triggered. Furthermore, assume that the GPIB device requests service when it has acquired a reading. Given that, the following steps are necessary:

Step 4a.

Program the GPIB device to acquire a reading whenever it receives a GPIB trigger using the `Send` command. The buffer that you pass to `Send` is the command message that programs the device to behave properly.

Step 4b.

Trigger the device using the `Trigger` routine.

Step 4c.

Wait for the device to acquire the reading using the `WaitSRQ` routine.

Step 4d.

If the wait for the service request succeeded, read the serial poll status byte and verify that it indicates that the device obtained a good measurement using the `ReadStatusByte` routine.

Step 4e.

Read the measurement from the device using the `Receive` routine and record it in a list of device measurements.

Steps 4b through 4e should be repeated for each measurement you want to acquire.

Step 5. Place the Device Offline Before Exiting Your Application

Once you are finished accessing the GPIB device, take it offline using the `ibonl` function before you exit your application.

Language-Specific Programming Instructions

The following sections describe how to develop, compile, and link your Win32 GPIB applications using various programming languages.

Microsoft Visual C/C++ (Version 2.0 or Higher)

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

```
#include <windows.h>
#include "decl-32.h"
```

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
```

Borland C/C++ (Version 4.0 or Higher)

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

```
#include <windows.h>
#include "decl-32.h"
```

To compile and link a Win32 console application named `cprog` in a DOS shell, type the following on the command line:

```
bcc32 -w32 cprog.c borlandc_gpib-32.obj
```

Visual Basic (Version 4.0 or Higher)

With Visual Basic, you can access the NI-488 functions as subroutines, using the BASIC keyword `CALL` followed by the NI-488 function name, or you can access the NI-488 functions using the `il` set of functions. With some of the NI-488 functions and NI-488.2 subroutines (for example `ibrd` or `Receive`) the length of the string buffer is automatically calculated within the actual function or subroutine, which eliminates the need to pass in the length as an extra parameter. Refer to the online help or *NI-488.2M Function Reference Manual for Win32* for more information about function syntax for Visual Basic.

Before you run your Visual Basic application, include the files `niglobal.bas` and `vbib-32.bas` in your application project file.

Direct Entry with C

The following sections describe how to use direct entry with C.

gpib-32.dll Exports

`gpib-32.dll` exports pointers to the global variables and all of the NI-488.2 functions and subroutines. Pointers to the global variables (`ibsta`, `iberr`, `ibcnt`, and `ibcntl`) are accessible through these exported variables:

```
int *user_ibsta;
int *user_iberr;
int *user_ibcnt;
long *user_ibcntl;
```

Except for the functions `ibbna`, `ibfind`, `ibrdf`, and `ibwrtf`, all of the NI-488.2 function and subroutine names are exported from `gpib-32.dll`. What this means is that to use direct entry to access a particular function all you need to do to get a pointer to the exported function is to call `GetProcAddress` passing the name of the function as a parameter. The parameters that you use when you invoke the function are identical to those described in the online help and *NI-488.2M Function Reference Manual for Win32*.

These functions all require an argument that is a name. `ibbna` requires a board name, `ibfind` requires a board or device name, and `ibrdf` and `ibwrtf` take a file name. Because Windows NT supports both normal (8-bit) and Unicode (16-bit) characters, `gpib-32.dll` exports both normal and Unicode versions of these functions. Because Windows 95 does not support 16-bit wide characters, use only the 8-bit

ASCII versions, named `ibbnaA`, `ibfindA`, `ibrdfA`, and `ibwrtfA`. The Unicode versions are named `ibbnaW`, `ibfindW`, `ibrdfW` and `ibwrtfW`. You can use either the Unicode or ASCII versions of these functions with Windows NT, but only the ASCII versions with Windows 95.

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

Directly Accessing the gpib-32.dll Exports

Make sure that the following lines are included at the beginning of your application:

```
#ifdef __cplusplus
extern "C"{
#endif

#include <windows.h>
#include "decl-32.h"

#ifdef __cplusplus
}
#endif
```

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:

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

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

```
/* Pointers to NI-488.2 global status variables */
int *Pibsta;
int *Piberr;
long *Pibcntl;
static int(__stdcall *Pibdev)
       (int ud, int pad, int sad, int tmo, int eot,
        int eos);
static int(__stdcall *Pibonl)
       (int ud, int v);
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)) {

   /* Free the GPIB library */
   FreeLibrary(Gpib32Lib);
   printf("GetProcAddress failed.");
}
```

Your Win32 application needs to dereference 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 examples of directly accessing `gpib-32.dll`, refer to the NI-488.2M direct entry sample programs `dlldev.c` and `dll4882.c` that are installed with the GPIB software. For more information on direct entry, refer to the Win32 SDK (Software Development Kit) online help.

Windows 95: Running Existing GPIB Applications

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 installed 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.

Running Existing DOS GPIB Applications

With the NI-488.2M software properly configured, you can run your existing DOS GPIB applications along with your Win16 and Win32 GPIB applications. No DOS device driver is required. In fact, be sure that no older version of the GPIB DOS device driver is being loaded from your `config.sys` file, a file located on the boot drive of your computer. The older GPIB DOS device driver is loaded with a command line of the form `device=<path>/gpib.com` where `<path>` is the drive and directory where `gpib.com` is located. Delete this line to ensure that the older GPIB DOS driver is not being loaded.

To run DOS GPIB applications, the system uses a Virtual Device Driver (VxD), `gpibdosk.vxd`, and a Win32 executable, `gpibdos.exe`. When you install the NI-488.2M software, `gpibdosk.vxd` and `gpibdos.exe` are copied into the Windows System directory. These files are loaded when you restart your computer, if the NI-488.2M software has been properly configured to run your existing DOS GPIB applications.

To configure the NI-488.2M software to run your existing DOS GPIB applications, follow these steps after you have installed the NI-488.2M software and your GPIB hardware:

1. Select **Start»Settings»Control Panel**, and double-click the **System** icon. The **System Properties** dialog box appears.
2. Select the **Device Manager** tab.
3. Click the **View devices by type** radio button at the top of the page, and click on the **National Instruments GPIB Interfaces** icon.
4. Click on the **Properties** button to display the **General** property page for the NI-488.2M software.
5. Select the checkbox labeled **Enable Support for DOS GPIB Applications**, and then click on the **OK** button.
6. Restart your computer.

Now you can run your existing DOS GPIB applications.

Windows NT: Running Existing GPIB Applications

You can run existing DOS and Windows GPIB applications under Windows NT by using the GPIB Virtual Device Driver, `gpib-vdd.dll`, which is included with your NI-488.2M software.

To run DOS GPIB applications, load the special GPIB device driver `gpib-nt.com` instead of `gpib.com`, which you normally use with DOS. When you install the NI-488.2M software, `gpib-nt.com` is copied into a new subdirectory called `doswin16`. To use `gpib-nt.com`, you must modify your `config.nt` file to load `gpib-nt.com` whenever a DOS application is executed. The `config.nt` file is located in your `<winnt>\system32` directory, where *`<winnt>`* is your Windows NT directory, for example `c:\windows`. To load `gpib-nt.com`, add the following line to your `config.nt` file:

```
device=<path>\doswin16\gpib-nt.com
```

where *`<path>`* is the directory where you installed the GPIB software (the default installation directory is `c:\gpib-nt`).

To run Win16 GPIB applications, the system uses the special GPIB dynamic link library, `gpib.dll`. When you install the NI-488.2M software, `gpib.dll` is copied into the *<winnt>*`\system32` directory, where *<winnt>* is your Windows NT directory, for example `c:\windows`). As long as `gpib.dll` is in that directory, it is automatically accessed whenever you launch a Win16 GPIB application.

Chapter 4

Debugging Your Application

This chapter describes several ways to debug your application.

Debugging with GPIB Spy

You can use the GPIB Spy utility to monitor all of the GPIB calls that are made by GPIB applications. Because all applications go through `gpib-32.dll`, the GPIB calls made by Win32, Win16, and DOS applications are all recorded by GPIB Spy. For more information about GPIB Spy, refer to the online help available through the application or to Chapter 5, *GPIB Spy Utility*.

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. 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 or Chapter 6, *Win32 Interactive Control Utility*.

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

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, 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 application uses dynamic configuration, it will always work regardless of the previous configuration of the driver. Refer to the description of `ibconfig` in the online help or 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 8, *GPIB Configuration Utility*, or to the description of `ibconfig` (option `IbcREADDR`) in the online help or *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);
```

Other Errors

If you experience other errors in your application, refer to Appendix C, *Windows 95: Troubleshooting and Common Questions*, or Appendix D, *Windows NT: Troubleshooting and Common Questions*, depending on which operating system you are using.

Chapter

5

GPIB Spy Utility

This chapter introduces you to GPIB Spy, the application monitor you can use to monitor NI-488 and NI-488.2 calls.

Overview

GPIB Spy monitors, records, and displays the NI-488 and NI-488.2 calls made to the NI-488.2M driver. GPIB Spy monitors Win32, Win16, and DOS GPIB applications. It is a useful tool for troubleshooting errors in your application and for verifying that the communication with your GPIB instrument is correct.

Starting GPIB Spy

When you launch GPIB Spy, it displays the main GPIB Spy window. By default, Spy capture is on, and GPIB Spy records all GPIB calls made to the NI-488.2M driver. Figure 5-1 shows the main GPIB Spy window with several calls recorded in it.

Starting GPIB Spy Under Windows 95

Start GPIB Spy by choosing the **GPIB Spy** item under **Start»Programs»NI-488.2M Software for Windows 95**.

Starting GPIB Spy Under Windows NT

In Windows NT 3.51, start GPIB Spy by double-clicking on the **GPIB Spy** icon in the **NI-488.2M Software for Windows NT** group of the **Program Manager**. In Windows NT 4.0 or later, start GPIB Spy by choosing the **GPIB Spy** item under **Start»Programs»NI-488.2M Software for Windows NT**.

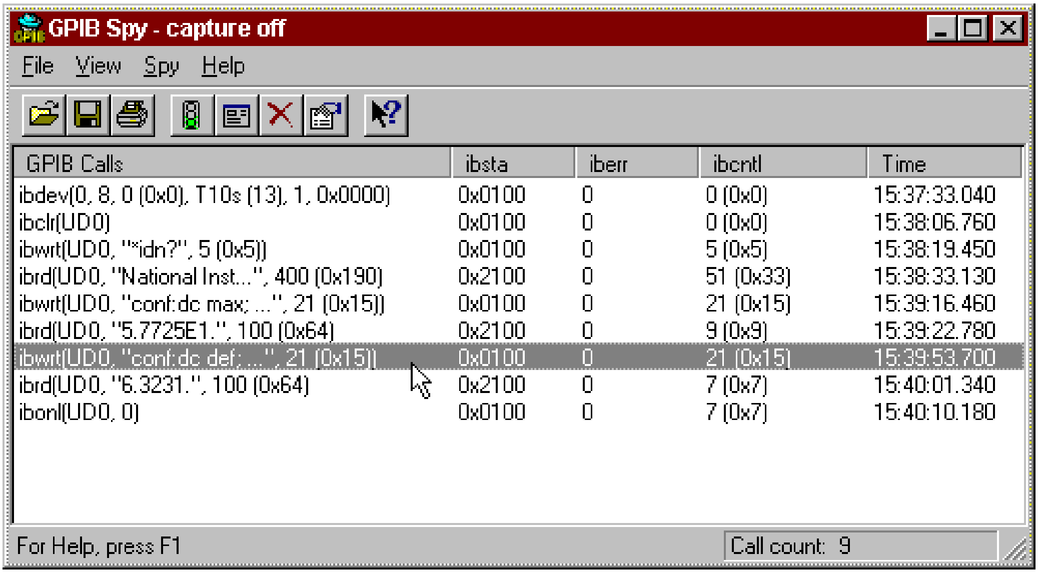


Figure 5-1. GPIB Spy Main Window

Using the Online GPIB Spy Help

The GPIB Spy utility has built-in, context-sensitive online help. You can access it through GPIB Spy's **Help** menu to view descriptions of all GPIB Spy features. You can also access the GPIB Spy context-sensitive help by clicking on the question mark button, and then clicking on any area of the screen.

Locating Errors with GPIB Spy

All GPIB calls returned with an error are displayed in red within the main GPIB Spy window.

Viewing Properties for Recorded Calls

You can view the detailed properties of any call recorded in the main GPIB Spy window by double-clicking on the call. The **Call Properties** window contains general, input, and output information. Figure 5-2 shows the **Input** tab for a device-level `ibwrt` and Figure 5-3 shows the **Output** tab for a device-level `ibrd` call.

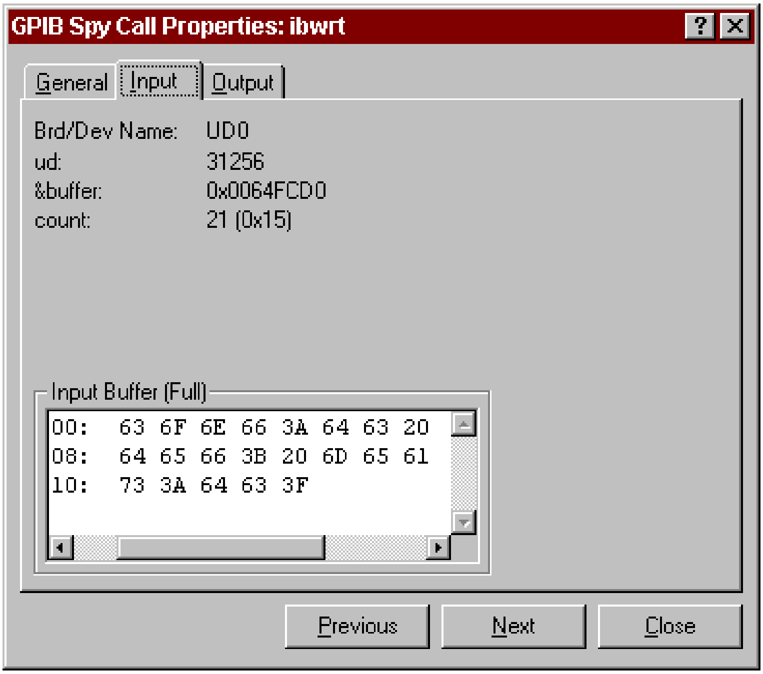


Figure 5-2. GPIB Spy Input Tab for Device-Level `ibwrt`

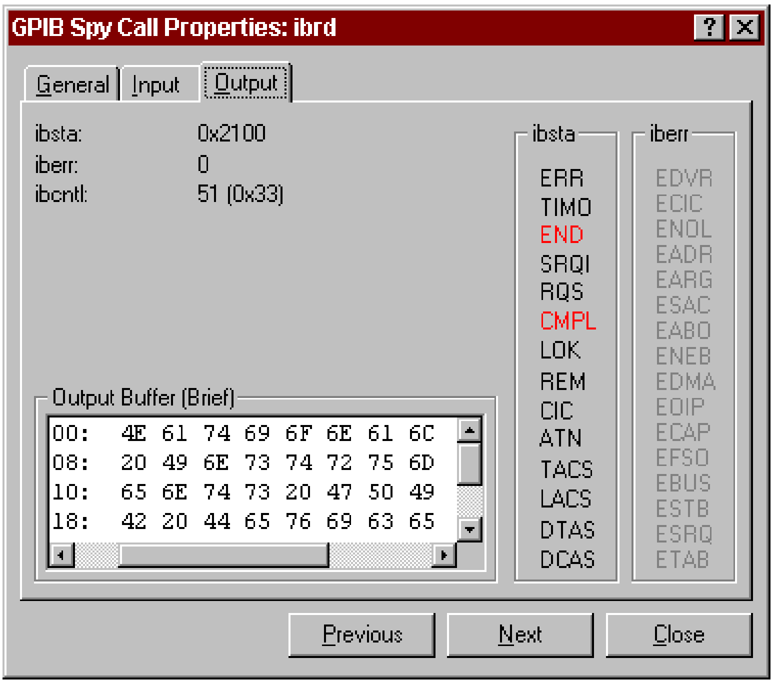


Figure 5-3. GPIB Spy Output Tab for Device-Level `ibrd`

Exiting GPIB Spy

When you exit GPIB Spy, its current configuration is saved and used to configure GPIB Spy when you start it again. Note that unless you explicitly save the data captured in GPIB Spy before you exit, that information is lost.

Performance Considerations

GPIB Spy can slow down the performance of your GPIB application, and certain configurations of GPIB Spy have a larger impact on performance than others. For example, configuring GPIB Spy to record calls to an output file or to use full buffers, might have a significant impact on the performance of both your application and the system. For this reason, use GPIB Spy only while you are debugging your application or in situations where performance is not critical.

Chapter 6

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.

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

Getting Started with Win32 Interactive Control

This section shows how you might use Win32 Interactive Control to test a sequence of GPIB calls.

Run the Win32 Interactive Control utility:

- Windows 95: Select the **Win32 Interactive Control** item under **Start»Programs»NI-488.2M Software for Windows 95**.

- Windows NT 3.51: Double-click on the **Win32 Interactive Control** icon in the **NI-488.2M Software for Windows NT** group of the Program Manager.
- Windows NT 4.0 and later: Select the **Win32 Interactive Control** item under **Start»Programs»NI-488.2M Software for Windows NT**.

When the Win32 Interactive Control utility first starts, it displays the following banner message:

```
Win32 Interactive Control
Copyright 1996 National Instruments Corporation
All rights reserved

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

:
```

The first step is to open either a board handle or device handle to use for further GPIB calls. Use `ibdev` to open a device handle, use `ibfind` to open a board handle or use the `set 488.2` command to switch to a 488.2 prompt. For help on any Win32 Interactive Control command, type in `help` followed by the command, for example, `help ibdev` or `help set`.

If you want to use device-level calls, open a device handle using `ibdev`. The following example shows how you can 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 seconds, enable the END message and disable the EOS mode;

```
:ibdev
   enter board index: 0
   enter primary address: 6
   enter secondary address: 0
   enter timeout: T10s
   enter 'EOI on last byte' flag: 1
   enter end-of-string mode/byte: 0
ud0:
```

If you enter a command and no parameters, you are automatically prompted for the necessary arguments. If you already know the required arguments, you can enter them from the command line, as follows:

```
:ibdev 0 6 0 T10s 1 0
ud0:
```

The new prompt, `ud0`, represents a device-level handle that can be used for further GPIB calls. To clear the device, use `ibclr` as follows:

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

To write data to the device, use `ibwrt`. Make sure that you refer to the instrument user manual that came with your GPIB instrument for specific command messages.

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

or, equivalently:

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

To send a trigger, use `ibtrg` as follows:

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

To read data from your device, use `ibrd`. The data that is read from the instrument is displayed. For example, to read 18 bytes:

```
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, equivalently:

```
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                        . .
```

When you are finished communicating with the device, make sure you put it offline using the `ibonl` command as follows:

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

This properly closes the device handle and the `ud0` prompt is no longer present.

Win32 Interactive Control Syntax

The following special rules apply to making calls from the interactive control utility.

- The `ud` or `BoardId` parameter is implied by the Interactive Control prompt, therefore it is never included in the call.
- The `count` parameter to functions is unnecessary because buffer lengths are automatically determined by the interactive control utility.
- Function return values are handled automatically by the interactive control utility. In addition to printing out the return `ibsta` value for the function, it also prints other return values.
- If you do not know what parameters are appropriate to pass to a given function call, type in the function name and press <Enter>, and the interactive control utility automatically prompts you for each required parameter.

Number Syntax

You can enter numbers as hexadecimal or decimal integer.

Hexadecimal numbers–You must precede hex numbers by zero and x (for example, 0xD).

Decimal numbers–Enter the number only.

String Syntax

You can enter strings as an ASCII character sequence, hex bytes, or special symbols.

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

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. An address list is represented by a comma-separated list of addresses such as `1,2,3`.

Win32 Interactive Control Commands

Tables 6-1 and 6-2 summarize the syntax of NI-488 functions in Win32 Interactive Control. Table 6-3 summarizes the syntax of NI-488.2 routines in Win32 Interactive Control. Table 6-4 summarizes the auxiliary functions that you can use in Win32 Interactive Control. For more information about the function parameters, use the online help feature. If you enter only the function name, the Win32 Interactive Control utility prompts you for parameters.

Table 6-1. Syntax for Device-Level NI-488 Functions in Win32 Interactive Control

Syntax	Description
`ibask option`	Return configuration information where `option` is a mnemonic for a configuration parameter
`ibbna bname`	Change access board of device where `bname` is symbolic name of new board
`ibclr`	Clear specified device
`ibconfig option value`	Alter configurable parameters where `option` is mnemonic for a configuration parameter
`ibdev BdIndx pad sad tmo eot eos`	Open an unused device `ibdev` parameters are `BdIndx pad sad tmo eot eos`
`ibeos v`	Change/disable EOS message
`ibeot v`	Enable/disable END message
`ibln pad sad`	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 count`	Read data where `count` is the bytes to read
`ibrda count`	Read data asynchronously where `count` 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 or decimal integer or a list of mask bit mnemonics such as `ibwait TIMO CMPL`
`ibwrt wrtbuf`	Write data
`ibwrta wrtbuf`	Write data asynchronously
`ibwrtf flname`	Write data from a file where `flname` is pathname of file to write

Table 6-2. Syntax for Board-Level NI-488 Functions in Win32 Interactive Control

Syntax	Description
`ibask option`	Return configuration information where `option` is a mnemonic for a configuration parameter
`ibcac v`	Become active Controller
`ibcmd cmdbuf`	Send commands
`ibcmda cmdbuf`	Send commands asynchronously
`ibconfig option value`	Alter configurable parameters where `option` is mnemonic for a configuration parameter
`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 pad sad`	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 count`	Read data where `count` is the bytes to read
`ibrda count`	Read data asynchronously where `count` 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 or decimal integer or a list of mask bit mnemonics such as `ibwait TIMO CMPL`
`ibwrt wrtbuf`	Write data
`ibwrta wrtbuf`	Write data asynchronously
`ibwrtf flname`	Write data from a file where `flname` is pathname of file to write

Table 6-3. 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 padlist limit`	Find all Listeners
`FindRQS addrlist`	Find device asserting SRQ
`PassControl address`	Pass control to a device
`PPoll`	Parallel poll devices
`PPollConfig address dataline lineSense`	Configure device for parallel poll
`PPollUnconfig addrlist`	Unconfigure device for parallel poll
`RcvRespMsg count termination`	Receive response message
`ReadStatusByte address`	Serial poll a device
`Receive address count termination`	Receive data from a device
`ReceiveSetup address`	Receive setup
`ResetSys addrlist`	Reset multiple devices
`Send address buffer eotmode`	Send data to a device
`SendCmds buffer`	Send command bytes
`SendDataBytes buffer eotmode`	Send data bytes
`SendIFC`	Send interface clear
`SendList addrlist buffer eotmode`	Send data to multiple devices
`SendLLO`	Put devices in local lockout
`SendSetup addrlist`	Send setup
`SetRWLS addrlist`	Put devices in remote with lockout state
`TestSRQ`	Test for service request
`TestSys addrlist`	Cause multiple devices to perform self-tests
`Trigger address`	Trigger a device
`TriggerList addrlist`	Trigger multiple devices
`WaitSRQ`	Wait for service request

Table 6-4. 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`	Display Win32 interactive utility online help.
`help option`	Display help information on `option`, where `option` is any NI-488, NI-488.2, or auxiliary call, for example `help ibwrt` or `help set`.
`!`	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.
`buffer option`	Set type of display used for buffers. Valid options are `full`, `brief`, `ascii`, and `off`. Default is `full`.
`q`	Exit or quit.

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, showing that the 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 the device `ud0` represents 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 Information

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 online help or *NI-488.2M Function Reference Manual for Win32* for the correct interpretation of the count return.

GPIB Programming Techniques

Chapter 7

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

For more detailed information about each function or routine, refer to the online help or *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 `IbcEndBitIsNormal` option to configure the software to report only the END bit in `ibsta` when the GPIB EOI line is asserted. By default, the NI-488.2M 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. 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. For CFE and CFGn messages, refer to the online help or 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.

Asynchronous Event Notification in Win32 GPIB Applications

Win32 GPIB applications can asynchronously receive event notifications using the `ibnotify` function. This function is useful if you want your application to be notified asynchronously about the occurrence of one or more GPIB events. For example, you might choose to use `ibnotify` if your application only needs to interact with your GPIB device when it is requesting service. After calling `ibnotify`, your application does not need to check the status of your GPIB device. Then when your GPIB device requests service, the GPIB driver automatically notifies your application that the event has occurred by invoking a callback function. The callback function is registered with the GPIB driver when the `ibnotify` call is made.

Calling the ibnotify Function

`ibnotify` has the following function prototype:

```
ibnotify (
    int ud,          // unit descriptor
    int mask,        // bit mask of GPIB events
    GpibNotifyCallback_t Callback,
                     // callback function
    void * RefData   // user-defined reference data
    )
```

Both board-level and device-level `ibnotify` calls are supported by the GPIB driver. If you are using device-level calls, you call `ibnotify` with a device handle for `ud` and a `mask` of RQS, CMPL, END, or TIMO. If you are using board-level calls, you call `ibnotify` with a board handle for `ud` and a `mask` of any values except RQS or ERR. Note that the `ibnotify` mask bits are identical to the `ibwait` mask bits. In the example of waiting for your GPIB device to request service, you might choose to pass `ibnotify` a `mask` with RQS (for device-level) or SRQI (for board-level).

The Callback function that you register with the `ibnotify` call is invoked by the GPIB driver when one or more of the mask bits passed to `ibnotify` is TRUE. The function prototype of the callback is as follows:

```
int __stdcall Callback (
    int ud,           // unit descriptor
    int ibsta,        // ibsta value
    int iberr,        // iberr value
    long ibcntl,      // ibcntl value
    void * RefData    // user-defined reference data
)
```

The callback function is passed a unit descriptor, the current values of the GPIB global variables, and the user-defined reference data that was passed to the original `ibnotify` call. The GPIB driver interprets the return value for the callback as a mask value that is used to automatically rearm the callback if it is non-zero. For a complete description of `ibnotify`, refer to the online help or *NI-488.2M Function Reference Manual for Win32*.

Note: ***The*** `ibnotify` ***Callback is executed in a separate thread of execution from the rest of your application. If your application might be performing other GPIB operations while it is using*** `ibnotify`***, you should use the per-thread GPIB globals that are provided by the*** `ThreadIbsta`, `ThreadIberr`, `ThreadIbcnt`, ***and*** `ThreadIbcntl` ***functions that are described in the*** *Writing Multithreaded Win32 GPIB Applications* ***section of this chapter. In addition, if your application needs to share global variables with the*** `Callback`***, you should use a synchronization primitive (for example, semaphore) to protect access to any globals. For more information on the use of synchronization primitives, refer to the documentation on using Win32 synchronization objects that came with your development tools.***

ibnotify Programming Example

The following code is an example of how you might use `ibnotify` in your application. Assume that your GPIB device is a multimeter that you program to acquire a reading by sending it “SEND DATA”. The multimeter requests service when it has a reading ready, and each reading is a floating point value.

In this example, globals are shared by the `Callback` thread and the main thread, and the access of the globals is not protected by synchronization. In this case, synchronization of access to these globals is not necessary because of the way they are used in the application: only a single thread is writing the global values and that thread always just adds information (increases the count or adds another reading to the array of floats).

```
int __stdcall MyCallback (int ud, int LocalIbsta, int LocalIberr,
                          long LocalIbcntl, void *RefData);
int ReadingsTaken = 0;
float Readings[1000];
BOOL DeviceError = FALSE;

int main()
{
   int ud;

   // Assign a unique identifier to the device and store it in the
   // variable ud. ibdev opens an available device and assigns it to
   // access GPIB0 with a primary address of 1, a secondary address of 0,
   // a timeout of 10 seconds, the END message enabled, and the EOS mode
   // disabled. If ud is less than zero, then print an error message
   // that the call failed and exit the program.
   ud = ibdev  (0,    // connect board
                1,    // primary address of GPIB device
                0,    // secondary address of GPIB device
                T10s, // 10 second I/O timeout
                1,    // EOT mode turned on
                0);   // EOS mode disabled

   if (ud < 0)  {
      printf ("ibdev failed.\n");
      return 0;
   }
```

```
    // Issue a request to the device to send the data. If the ERR bit
    // is set in ibsta, then print an error message that the call failed
    // and exit the program.
    ibwrt (ud, "SEND DATA", 9L);
    if (ibsta & ERR)  {
       printf ("unable to write to device.\n");
       return 0;
    }

    // set up the asynchronous event notification on RQS
    ibnotify (ud, RQS, MyCallback, NULL);
    if (ibsta & ERR)  {
       printf ("ibnotify call failed.\n");
       return 0;
    }

    while ((ReadingsTaken < 1000) && !(DeviceError))  {
       // Your application does useful work here. For example, it
       // might process the device readings or do any other useful work.
    }

    // disable notification
    ibnotify (ud, 0, NULL, NULL);

    // Call the ibonl function to disable the hardware and software.
    ibonl (ud, 0);
    return 1;

}

int __stdcall MyCallback (int LocalUd, int LocalIbsta, int LocalIberr,
                          long LocalIbcntl, void *RefData)
{
    char SpollByte;
    char ReadBuffer[40];

    // If the ERR bit is set in LocalIbsta, then print an error message
    // and return.
    if (LocalIbsta & ERR)  {
       printf ("GPIB error %d has occurred. No more callbacks.\n",
               LocalIberr);
       DeviceError = TRUE;
       return 0;
    }
```

```
// Read the serial poll byte from the device. If the ERR bit is set
// in ibsta, then print an error message and return.
LocalIbsta = ibrsp (LocalUd, &SpollByte);
if (LocalIbsta & ERR)  {
   printf ("ibrsp failed. No more callbacks.\n");
   DeviceError = TRUE;
   return 0;
}

// If the returned status byte equals the expected response, then
// the device has valid data to send; otherwise it has a fault
// condition to report.
if (spr != expectedResponse)   {
   printf("Device returned invalid response. Status byte = 0x%x\n",
          spr);
   DeviceError = TRUE;
   return 0;
}

// Read the data from the device. If the ERR bit is set in ibsta,
// then print an error message and return.
LocalIbsta = ibrd (LocalUd, ReadBuffer, 40L);
if (LocalIbsta & ERR)  {
   printf ("ibrd failed. No more callbacks.\n");
   DeviceError = TRUE;
   return 0;
}

// Convert the data into a numeric value.
sscanf (ReadBuffer, "%f", &Readings[ReadingsTaken]);
ReadingsTaken += 1;
if (ReadingsTaken >= 1000)  {
   return 0;
}

else  {
   // Issue a request to the device to send the data and rearm
   // callback on RQS.
   LocalIbsta = ibwrt (LocalUd, "SEND DATA", 9L);
   if (LocalIbsta & ERR)  {
      printf ("ibwrt failed. No more callbacks.\n");
      DeviceError = TRUE;
      return 0;
   }
```

```
        else  {
            return RQS;
        }
    }
}
```

Writing Multithreaded Win32 GPIB Applications

If you are writing a multithreaded GPIB application and you plan to make all of your GPIB calls from a single thread, you can safely continue to use the traditional GPIB global variables (`ibsta`, `iberr`, `ibcnt`, `ibcntl`). The GPIB global variables are defined on a per-process basis, so each process accesses its own copy of the GPIB globals.

If you are writing a multithreaded GPIB application and you plan to make GPIB calls from more than a single thread, you cannot safely continue to use the traditional GPIB global variables without some form of synchronization (for example, a semaphore). To understand why this is true, take a look at the following example.

Assume that a process has two separate threads that make GPIB calls, thread #1 and thread #2. Just as thread #1 is about to examine one of the GPIB globals, it gets preempted and thread #2 is allowed to run. Thread #2 proceeds to make several GPIB calls that automatically update the GPIB globals. Later, when thread #1 is allowed to run, the GPIB global that it is ready to examine is no longer in a known state and its value is no longer reliable.

This example illustrates a well-known multithreading problem. It is unsafe to access process-global variables from multiple threads of execution. You can avoid this problem in two ways:

- Use synchronization to protect access to process-global variables.
- Do not use process-global variables.

If you choose to implement the synchronization solution, you must ensure that code that makes GPIB calls and examines the GPIB globals modified by a GPIB call is protected by a synchronization primitive. For example, each thread might acquire a semaphore before making a GPIB call and then release the semaphore after examining the GPIB globals modified by the call. For more information on the use of synchronization primitives, refer to the documentation on using Win32 synchronization objects that came with your development tools.

If you choose not to use process-global variables, you can access per-thread copies of the GPIB global variables using a special set of GPIB calls. Whenever a thread makes a GPIB call, the driver keeps a private copy of the GPIB globals for that thread. The driver keeps a separate private copy for each thread. The following code shows the set of functions you can use to access these per-thread GPIB global variables.

```
int ThreadIbsta();   // return thread-specific ibsta
int ThreadIberr();   // return thread-specific iberr
int ThreadIbcnt();   // return thread-specific ibcnt
long ThreadIbcntl(); // return thread-specific ibcntl
```

In your application, instead of accessing the per-process GPIB globals, substitute a call to get the corresponding per-thread GPIB global. For example, the line of code

```
if (ibsta & ERR)
```

could be replaced by

```
if (ThreadIbsta() & ERR)
```

A quick way to convert your application to use per-thread GPIB globals, is to add the following `#define` lines at the top of your C file:

```
#define ibsta       ThreadIbsta()
#define iberr       ThreadIberr()
#define ibcnt       ThreadIbcnt()
#define ibcntl      ThreadIbcntl()
```

Note: ***If you are using*** `ibnotify` ***in your application (see the*** *Asynchronous Event Notification in Win32 GPIB Applications* ***section of this chapter) the*** `ibnotify` ***callback is executed in a separate thread that is created by the GPIB driver. Therefore, if your application makes GPIB calls from the*** `ibnotify` ***callback function and makes GPIB calls from other places, you must use the*** `ThreadIbsta`, `ThreadIberr`, `ThreadIbcnt`, ***and*** `ThreadIbcntl` ***functions described in this section, instead of the per process GPIB globals***.

Device-Level Calls and Bus Management

The NI-488 device-level calls are designed to perform all of the GPIB management for your application. 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 and Windows NT 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. If automatic serial polling is enabled, the application 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 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 8

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 Windows 95 GPIB configuration utility is integrated into the Windows 95 Device Manager. The Windows NT GPIB configuration utility is integrated into the Windows NT Control Panel. You can use the GPIB configuration utility 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. If your application 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.

Windows 95: Configuring 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**. If an exclamation point or an X appears next to the interface, there is a problem, and you should refer to the *Troubleshooting Windows 95 Device Manager Device Status Codes* section of Appendix C, *Windows 95: Troubleshooting and Common Questions*, to resolve your problem before you continue. 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 8-1 shows the **NI-488.2M Settings** tab for an AT-GPIB/TNT (PnP).

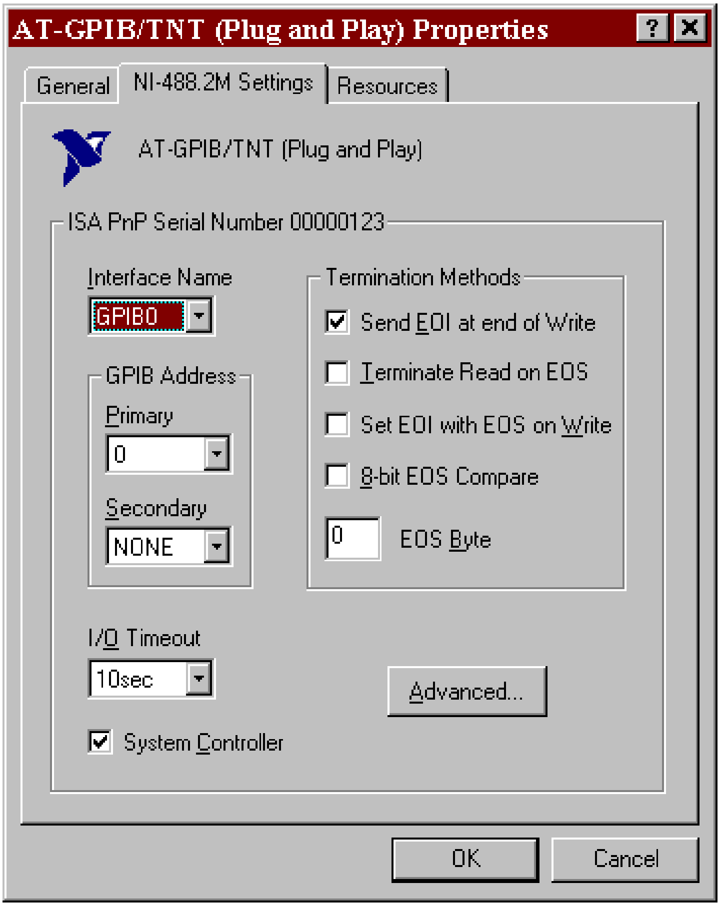


Figure 8-1. NI-488.2M Settings Tab for the AT-GPIB/TNT (PnP)

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 8-2.

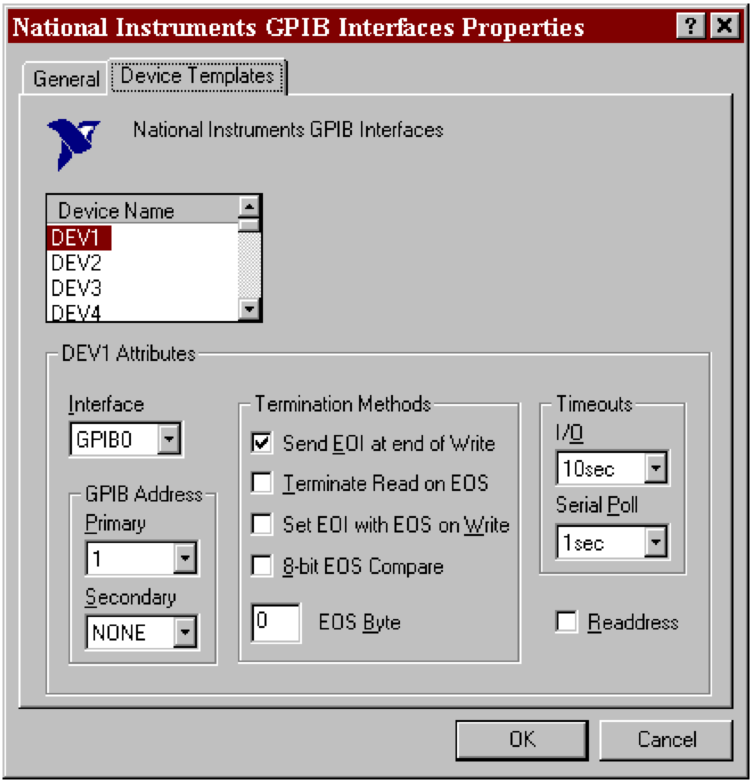


Figure 8-2. Device Templates Tab for the Logical Device Templates

Windows NT: Configuring the NI-488.2M Software

When you install the NI-488.2M software for Windows NT, the installation program places the GPIB configuration utility into your Control Panel. To start the GPIB configuration utility simply open your Windows NT Control Panel and select the eagle icon.

Because you can use the GPIB configuration utility to modify the configuration of the NI-488.2M kernel drivers, you must be logged on to Windows NT as the `Administrator` to make any changes with the GPIB configuration utility. If you start the GPIB configuration utility without `Administrator` privileges, it runs in read-only mode; you can view the settings, but you cannot make changes.

The main **GPIB Configuration** dialog box appears containing a list of the GPIB boards and device templates as shown in Figure 8-3.

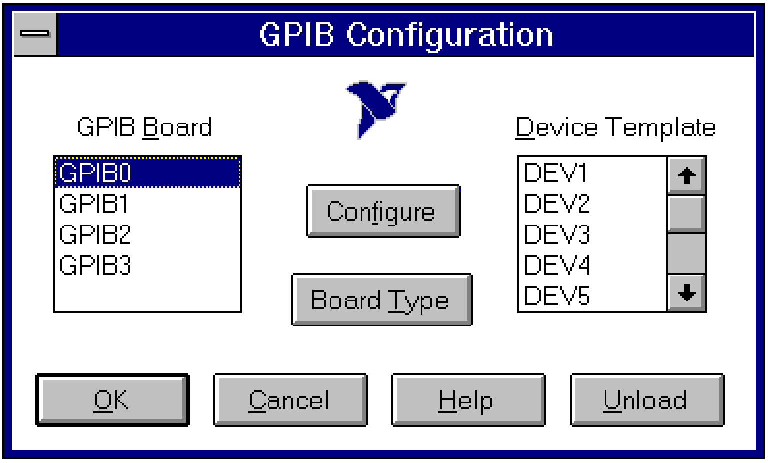


Figure 8-3. Main Dialog Box in the GPIB Configuration Utility

If at any point you need more help, click on the **Help** button or press the <F1> key. Either of these actions brings up the help screen, which gives you more information about the current dialog box.

After you have finished configuring your GPIB boards and device templates, click on the **OK** button to save the changes and exit. Click the **Cancel** button to exit without saving any of the changes you made.

After you click on the **OK** button, the GPIB Configuration utility asks whether or not you want the changes to take effect immediately. If you answer **No**, you must restart your system before the new settings can be used. If you answer **Yes**, the GPIB configuration utility attempts to unload and reload the NI-488.2M software so that the software uses your new settings. If the GPIB configuration utility cannot unload the software because it is being used by another application, it instructs you to restart your computer.

If you need to unload the NI-488.2M software and prevent it from reloading when you restart your computer, click the **Unload** button. If the GPIB configuration utility cannot unload the NI-488.2M software, it instructs you either to exit all GPIB-related applications, or to shut down and restart your computer. If you want to use the software again after unloading it, run the GPIB Configuration utility again and then click on the **OK** button.

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*.

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` or `ibnotify` call if the TIMO bit of the 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 one or more positive serial poll response bytes have been received from the device. A positive serial poll response byte always has bit 6 asserted. Automatic serial polling must be enabled (it is enabled by default) for RQS to automatically appear in `ibsta`. You can also wait for a device to request service regardless of the state of automatic serial polling by calling `ibwait` with a mask that contains RQS. Do not issue an `ibwait` call on RQS for a device that does not respond to serial polls. Use `ibrsp` to acquire the serial poll response byte that was received. RQS is cleared when all of the stored serial poll response bytes have been reported to you through the `ibrsp` function.

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.

If you use the `ibwait` or `ibnotify` function to wait for DCAS and the wait is completed, DCAS is cleared from `ibsta` after the next GPIB call. The same is true of reads and writes. If you call a read or write function such as `ibwrt` or `Send`, and DCAS is set in `ibsta`, the I/O operation is aborted. DCAS is cleared from `ibsta` after the next GPIB call.

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 an error code. 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 GPIB Configuration utility to ensure that each board you want to access is configured properly.
- 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.
- For Windows 95, refer to the *Troubleshooting EDVR Error Conditions* section in Appendix C, *Windows 95: Troubleshooting and Common Questions*, 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 if a system DMA error is encountered when the NI-488.2M software attempts to transfer data over the GPIB using DMA.

Solutions

- You can correct the EDMA problem in the hardware by using the GPIB configuration utility to reconfigure the hardware to not use a DMA resource.
- You can correct the EDMA problem in the software by using `ibdma` to disable DMA.

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`, `ibnotify`, `ibwait`, and `ibonl` or perform other non-GPIB operations. If any other call is attempted, EOIP is returned.

Solutions

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

- `ibnotify` If the `ibsta` value passed to the `ibnotify` callback contains CMPL, the driver and application are resynchronized.
- `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 operating system error code for this condition is contained in `ibcntl`.

Solutions

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

EBUS (14)

EBUS results when certain GPIB bus errors occur during device functions. All device functions send command bytes to perform addressing and other bus management. Devices are expected to accept these command bytes within the time limit specified by the default configuration or the `ibtmo` function. EBUS results if a timeout occurred while sending these command bytes.

Solutions

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

ESTB (15)

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

Solutions

- Call `ibrsp` more frequently to empty the queue.
- Disable autopolling with the `ibconfig` function (option `IbcAUTOPOLL`) or the GPIB configuration utility.

ESRQ (16)

ESRQ can only be returned by a device-level `ibwait` call with RQS set in the mask. 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 `ibsta` 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.

Windows 95: Troubleshooting and Common Questions

Appendix
C

This appendix describes how to troubleshoot problems and lists some common questions for Windows 95 users.

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 (-534249428)

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 (-535560155)

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 (-535560139)

If a call is made with a device name that is not listed in the logical device templates that are part of the NI-488.2M software configuration utility, an EDVR error condition occurs with `ibcntl` set to 0xE0140035.

EDVR Error with ibcntl Set to 0xE0320029 (-533594071)

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. This error is also returned if you try to access a PCMCIA-GPIB when 32-bit PCMCIA drivers are not enabled. Refer to the *Install the PCMCIA-GPIB+ or PCMCIA-GPIB* section in your getting started manual for more information about enabling 32-bit PCMCIA drivers.

EDVR Error with ibcntl Set to 0xE0140004 (-535560188)

This error may occur if the GPIB interface has not been correctly installed and detected by Windows 95. Refer to the *Installation and Configuration* chapter in your getting started manual for details on how to install the GPIB hardware. If you have already followed those instructions and still receive this error, Windows 95 might have configured the GPIB interface as an "Other Device". Refer to your getting started manual for information on how to force Windows 95 to detect the GPIB hardware.

Troubleshooting Windows 95 Device Manager Device Status Codes

If you are having trouble with your GPIB interface, check to see if the interface listing in the Windows 95 Device Manager appears with an exclamation point or X by it. If it does, click on the interface listing and then click on the **Properties** button to view the **General** properties page for the interface. In the **Device Status** section, look for the status description and status code number. Use these status code descriptions and numbers to troubleshoot your problem. The following paragraphs describe the status codes.

- Code 8: The GPIB software was incompletely installed. You might encounter this problem if you have installed an AT-GPIB/TNT+ but not installed the GPIB Analyzer software. To solve this problem, reinstall the GPIB software for Windows 95.
- Code 9: Windows 95 had a problem reading information from the GPIB interface. This problem can occur if you are using an older revision of the AT-GPIB/TNT+ or AT-GPIB/TNT (PnP) interface. Contact National Instruments to upgrade your GPIB interface.

- Code 15: The GPIB interface was not assigned an Interrupt Request level. If your computer does not have any available Interrupt Request levels, Windows 95 might configure your GPIB interface without an Interrupt Request level. The GPIB software for Windows 95 cannot function without an Interrupt Request level. Another way to verify this problem is to look at the **Resource settings** list on the **Resources** tab to verify that the GPIB interface was not assigned an Interrupt Request level. To solve this problem, free up an Interrupt Request level that is being used by another device in the system.
- Code 22: The GPIB interface is disabled. To enable the GPIB interface, check the appropriate configuration checkbox in the **Device Usage** section of the **General** tab.
- Code 24: The GPIB interface is not present, or the Device Manager is unaware that the GPIB interface is present. To solve this problem, select the interface in the Device Manager, and click on the **Remove** button. Next, click the **Refresh** button. At this point, the system rescans the installed hardware, and the GPIB interface should show up without any problems. If the problem persists, contact National Instruments.
- Code 27: Windows 95 was unable to assign the GPIB interface any resources. To solve this problem, free up system resources by disabling other unnecessary hardware so that enough resources are available for the GPIB interface.

Common Questions

What do I do if my GPIB hardware is listed in the Windows 95 Device Manager with an exclamation point or an X next to it?

Refer to the *Troubleshooting Windows 95 Device Manager Device Status Codes* section of this appendix for specific information about what might cause this problem. If you have already completed the troubleshooting steps, fill out the forms in Appendix E, *Customer Communication*, and contact National Instruments.

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

Run the GPIB Configuration utility. To run the utility, select **Start»Settings»Control Panel»System**. Select the **Device Manager** tab in the **System Properties** dialog box. Click on the **View devices by type** radio button at the top of the page. If any GPIB hardware is correctly installed, a **National Instruments GPIB Interfaces** icon appears in the list of device types. Double-click this icon to see a list of installed GPIB hardware.

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

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

Which GPIB interfaces does version 1.1 of the NI-488.2M software for Windows 95 support?

Version 1.1 of the GPIB software for Windows 95 supports the AT-GPIB/TNT, AT-GPIB/TNT (Plug and Play), AT-GPIB/TNT+, PCMCIA-GPIB, PCMCIA-GPIB+, PCI-GPIB, GPIB-ENET, EISA-GPIB, NEC-GPIB/TNT, and NEC-GPIB/TNT (Plug and Play).

I have a GPIB interface that the NI-488.2M software for Windows 95 does not support. What should I do?

Contact National Instruments to order the Compatibility Release for Windows 95 or to upgrade your hardware.

How many GPIB interfaces can I configure for use with my NI-488.2M software for Windows 95?

The NI-488.2M software for Windows 95 can be configured to communicate with up to 100 GPIB interfaces.

How many devices can I configure for use with my NI-488.2M software for Windows 95?

The NI-488.2M software for Windows 95 provides a total of 1,024 logical devices for applications to use. The default number of devices is 32. The maximum number of physical devices you should connect to a single GPIB interface is 14, or fewer, depending on your system configuration.

Are interrupts and DMA required with the NI-488.2M software for Windows 95?

Interrupts are required, but DMA is not.

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

Run the Diagnostic utility. To run the utility, select the **Diagnostic** item under **Start»Programs»NI-488.2M Software for Windows 95**. Use the Diagnostic online help to troubleshoot problems.

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. For more information, refer to Chapter 6, *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*.

What do I do if the Diagnostic utility fails with an error?

Use the Diagnostic online help, or refer to the getting started manual, to troubleshoot specific problems. If you have already completed the troubleshooting steps, fill out the support forms in Appendix E, *Customer Communication*, and contact National Instruments.

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*.

Why does the uninstall program leave some components installed?

The uninstall program removes only items that the installation program installed. If you add anything to a directory that was created by the installation program, the uninstall program does not delete that directory, because the directory is not empty after the uninstallation. You can remove the remaining components yourself.

What information should I have before I call National Instruments?

When you call National Instruments, you should have the results of the diagnostic test. Also, make sure you have filled out the technical support form in Appendix E, *Customer Communication*.

Windows NT: Troubleshooting and Common Questions

Appendix D

This appendix describes how to troubleshoot problems and lists some common questions for Windows NT users.

Using Windows NT Diagnostic Tools

There are many reasons why the NI-488.2M driver might not load. If the software is not properly installed or if there is a conflict between the GPIB hardware and the other hardware in the system, the NI-488.2M driver fails to start. Two Windows NT utilities are useful in determining the source of the problem: the **Devices** applet in the **Control Panel**, and the **Event Viewer**. The information available through each utility is described in the following sections.

Examining NT Devices to Verify the NI-488.2M Installation

To verify whether the NI-488.2M devices are installed correctly (that is, that the devices are started), run the **Devices** applet in the **Control Panel**. In Windows NT 3.51, open the GPIB **Control Panel** in the **Main** group of the **Program Manager**. In Windows NT 4.0 or later, select **Start»Settings»Control Panel**. This utility lists all of the devices Windows NT detects. Each device has a status associated with it. If the NI-488.2M driver is installed correctly, the following lines appear in the list of NT devices:

Device	Status	Started
GPIB Board Class Driver	Started	Automatic
GPIB Device Class Driver	Started	Automatic

You should also see one or more lines similar to the following:

Device	Status	Started
GPIB Port Driver (AT-GPIB)	****	System
GPIB Port Driver (PCI-GPIB)	****	System

The **GPIB Board Class Driver** and the **GPIB Device Class Driver** should both have a status of **Started**. If not, refer to the next section, *Examining the NT System Log Using the Event Viewer*.

At least one of the **GPIB Port Drivers** listed by the **Devices** applet should have a status of **`Started`**. If not, refer to the next section, *Examining the NT System Log Using the Event Viewer*.

If the **GPIB Class Driver** lines are not present or at least one **GPIB Port Driver** line is not present, the NI-488.2M software is not installed properly. You must reinstall the NI-488.2M software.

Examining the NT System Log Using the Event Viewer

Windows NT maintains a system log. If the NI-488.2M driver is unable to start, it records entries in the system log explaining why it failed to start. You can examine the system log by running the **Event Viewer** utility. In Windows NT 3.51, double-click on the Event Viewer icon in the **Administrative Tools** group of the **Program Manager**. In Windows NT 4.0 or higher, select **Start»Programs»Administrative Tools»Event Viewer**. Events that might appear in the system log include the following:

- The system is unable to locate the device file for one or more of the devices that make up the NI-488.2M driver and an event is logged that **`The system cannot find the file specified.`** In this case, the NI-488.2M software is incorrectly installed. You should reinstall the software.
- A conflict exists between the GPIB hardware and the other hardware in the system. If this is the case, an event is logged that indicates the nature of the resource conflict. To correct this conflict, reconfigure the GPIB hardware and NI-488.2M software. Refer to the getting started manual for configuration information.

Common Questions

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

Run the GPIB Configuration utility. To run the utility, open your Windows NT Control Panel and select the National Instruments eagle icon.

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

Run the Diagnostic utility. In Windows NT version 3.51, start the Diagnostic by double-clicking on the **Diagnostic** icon in the **NI-488.2M Software for Windows NT** group of the **Program Manager**. In Windows NT version 4.0 or later, start the Diagnostic by choosing the **Diagnostic** item under **Start»Programs»NI-488.2M Software for Windows NT**.

Which GPIB interfaces does version 1.2 of the NI-488.2M Software for Windows NT support?

Version 1.2 of the NI-488.2M Software for Windows NT supports the AT-GPIB, AT-GPIB/TNT, PCMCIA-GPIB, PCMCIA-GPIB+, and PCI-GPIB.

How many GPIB interfaces can I configure for use with my NI-488.2M Software for Windows NT?

The NI-488.2M Software for Windows NT can be configured to communicate with up to 4 GPIB interfaces.

How many devices can I configure for use with my NI-488.2M Software for Windows NT?

The NI-488.2M Software for Windows NT provides a total of 100 logical devices for applications to use. The default number of devices is 32.

Are interrupts and DMA required with the NI-488.2M Software for Windows NT?

Interrupts are required, but DMA is not.

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

Run the Diagnostic utility. In Windows NT version 3.51, start the Diagnostic by double-clicking on the **Diagnostic** icon in the **NI-488.2M Software for Windows NT** group of the **Program Manager**. In Windows NT version 4.0 or later, start the Diagnostic by choosing the **Diagnostic** item under **Start»Programs»NI-488.2M Software for Windows NT**.

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. For more information, refer to Chapter 6, *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*.

What do I do if the Diagnostic utility fails with an error?

Use the Diagnostic online help, or refer to the getting started manual, to troubleshoot specific problems. If you have already completed the troubleshooting steps, fill out the support forms in Appendix E, *Customer Communication*, and contact National Instruments.

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 Diagnostic test. Also, make sure you have filled out the technical support form in Appendix E, *Customer Communication*.

Appendix

E

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *Technical Support Form* before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

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 number:

(512) 418-1111

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`
LabWindows: `lw.support@natinst.com`
LabVIEW: `lv.support@natinst.com`
HiQ: `hiq.support@natinst.com`
VISA: `visa.support@natinst.com`

Telephone and Fax Support

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

	Telephone	Fax
Australia	03 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	
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
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 413091	02 41309215
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 and Windows NT

Edition Date: June 1996

Part Number: 321037B-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 (______) ______________________ Fax (______) ______________________

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*.
semaphore	An object that maintains a count between zero and some maximum value, limiting the number of threads that are simultaneously accessing a shared resource.
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

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W