NI-488.2™

NI-488.2 User Manual

June 2015
370428V-01

NATIONAL INSTRUMENTS™

Worldwide Technical Support and Product Information

ni.com

Worldwide Offices

Visit ni.com/niglobal to access the branch office websites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

For further support information, see the *NI Services* appendix. To comment on National Instruments documentation, refer to the National Instruments website at ni.com/info and enter the Info Code feedback.

Legal Information

Limited Warranty

This document is provided ‘as is’ and is subject to being changed, without notice, in future editions. For the latest version, refer to `ni.com/manuals`. NI reviews this document carefully for technical accuracy; however, NI MAKES NO EXPRESS OR IMPLIED WARRANTIES AS TO THE ACCURACY OF THE INFORMATION CONTAINED HEREIN AND SHALL NOT BE LIABLE FOR ANY ERRORS.

NI warrants that its hardware products will be free of defects in materials and workmanship that cause the product to fail to substantially conform to the applicable NI published specifications for one (1) year from the date of invoice.

For a period of ninety (90) days from the date of invoice, NI warrants that (i) its software products will perform substantially in accordance with the applicable documentation provided with the software and (ii) the software media will be free from defects in materials and workmanship.

If NI receives notice of a defect or non-conformance during the applicable warranty period, NI will, in its discretion: (i) repair or replace the affected product, or (ii) refund the fees paid for the affected product. Repaired or replaced Hardware will be warranted for the remainder of the original warranty period or ninety (90) days, whichever is longer. If NI elects to repair or replace the product, NI may use new or refurbished parts or products that are equivalent to new in performance and reliability and are at least functionally equivalent to the original part or product.

You must obtain an RMA number from NI before returning any product to NI. NI reserves the right to charge a fee for examining and testing Hardware not covered by the Limited Warranty.

This Limited Warranty does not apply if the defect of the product resulted from improper or inadequate maintenance, installation, repair, or calibration (performed by a party other than NI); unauthorized modification; improper environment; use of an improper hardware or software key; improper use or operation outside of the specification for the product; improper voltages; accident, abuse, or neglect; or a hazard such as lightning, flood, or other act of nature.

THE REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND THE CUSTOMER’S SOLE REMEDIES, AND SHALL APPLY EVEN IF SUCH REMEDIES FAIL OF THEIR ESSENTIAL PURPOSE.

EXCEPT AS EXPRESSLY SET FORTH HEREIN, PRODUCTS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND AND NI DISCLAIMS ALL WARRANTIES, EXPRESSED OR IMPLIED, WITH RESPECT TO THE PRODUCTS, INCLUDING ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE OR NON-INFRINGEMENT, AND ANY WARRANTIES THAT MAY ARISE FROM USAGE OF TRADE OR COURSE OF DEALING. NI DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF OR THE RESULTS OF THE USE OF THE PRODUCTS IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NI DOES NOT WARRANT THAT THE OPERATION OF THE PRODUCTS WILL BE UNINTERRUPTED OR ERROR FREE.

In the event that you and NI have a separate signed written agreement with warranty terms covering the products, then the warranty terms in the separate agreement shall control.

Copyright

End-User License Agreements and Third-Party Legal Notices

You can find end-user license agreements (EULAs) and third-party legal notices in the following locations:

- Notices are located in the `<National Instruments>\_Legal Information` and `<National Instruments>` directories.
- EULAs are located in the `<National Instruments>\Shared\MDF\Legal\license` directory.
- Review `<National Instruments>\_Legal Information.txt` for information on including legal information in installers built with NI products.

U.S. Government Restricted Rights

If you are an agency, department, or other entity of the United States Government (“Government”), the use, duplication, reproduction, release, modification, disclosure or transfer of the technical data included in this manual is governed by the Restricted Rights provisions under Federal Acquisition Regulation 52.227-14 for civilian agencies and Defense Federal Acquisition Regulation Supplement Section 252.227-7014 and 252.227-7015 for military agencies.

Trademarks

Refer to the *NI Trademarks and Logo Guidelines* at `ni.com/trademarks` for more information on National Instruments trademarks.

ARM, Keil, and µVision are trademarks or registered of ARM Ltd or its subsidiaries.

LEGO, the LEGO logo, WEDO, and MINDSTORMS are trademarks of the LEGO Group.

TETRIX by Pitsco is a trademark of Pitsco, Inc.

FIELDBUS FOUNDATION™ and FOUNDATION™ are trademarks of the Fieldbus Foundation.

Patents

For patents covering National Instruments products/technology, refer to the appropriate location: **Help»Patents** in your software, the patents.txt file on your media, or the *National Instruments Patent Notice* at ni.com/patents.

Export Compliance Information

Refer to the *Export Compliance Information* at ni.com/legal/export-compliance for the National Instruments global trade compliance policy and how to obtain relevant HTS codes, ECCNs, and other import/export data.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

YOU ARE ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY AND RELIABILITY OF THE PRODUCTS WHENEVER THE PRODUCTS ARE INCORPORATED IN YOUR SYSTEM OR APPLICATION, INCLUDING THE APPROPRIATE DESIGN, PROCESS, AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

PRODUCTS ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING IN THE OPERATION OF NUCLEAR FACILITIES; AIRCRAFT NAVIGATION; AIR TRAFFIC CONTROL SYSTEMS; LIFE SAVING OR LIFE SUSTAINING SYSTEMS OR SUCH OTHER MEDICAL DEVICES; OR ANY OTHER APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, “HIGH-RISK USES”). FURTHER, PRUDENT STEPS MUST BE TAKEN TO PROTECT AGAINST FAILURES, INCLUDING PROVIDING BACK-UP AND SHUT-DOWN MECHANISMS. NI EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES.

Contents

Chapter 3
GPIB Explorer (OS X and Linux)

Chapter 4
Developing Your NI-488.2 Application

Chapter 5
Debugging Your Application

Chapter 6
NI I/O Trace Utility

Chapter 7
Interactive Control Utility

Chapter 8
NI-488.2 Programming Techniques

Appendix A
GPIB Basics

Appendix B
Status Word Conditions

Appendix C
Error Codes and Solutions

Appendix D
Common Questions

Appendix E
NI Services

Glossary

Index

About This Manual

This manual describes the features and functions of the NI-488.2 software. You can use the NI-488.2 software with Windows, OS X, and Linux. Refer to the general readme file located on your installation media or in the installation directory, for the operating system versions supported by the current version of NI-488.2.

Using the NI-488.2 Documentation

The following NI-488.2 documentation is available with your NI-488.2 software distribution media:

- The *Getting Started*/*Installation Guide* briefly describes how to install the NI-488.2 software and your GPIB hardware.
- This manual describes the features and functionality of the NI-488.2 software.
- The *GPIB Hardware Installation Guide and Specifications* contains detailed instructions on how to install and configure your GPIB hardware. This guide also includes hardware specifications and compliance information.

To view these documents, you need Adobe Acrobat Reader, which you can download from `www.adobe.com`.

Windows

To view these documents, insert your NI-488.2 software distribution media and select the **View Documentation** option. The View Documentation utility helps you find the documentation that you want to view. You can also view these documents at `ni.com`.

OS X

To view these documents, insert your NI-488.2 software distribution media and open the **Documentation** folder. You can also view these documents at `ni.com`.

Linux

To view these documents, insert your NI-488.2 software distribution media and browse to the **Documentation** directory. You can also view these documents at `ni.com`.

Accessing the NI-488.2 Help

The *NI-488.2 Help* addresses questions you might have about NI-488.2 and includes a function reference and troubleshooting information.

Windows

Select **Start»Programs»National Instruments»Measurement & Automation** to start Measurement & Automation Explorer (MAX). **(Windows 8)** Click **NI Launcher** and select **Measurement & Automation Explorer**. Select **Help»Help Topics»NI-488.2**.

OS X

Select **Applications»National Instruments»NI-488.2»GPIB Explorer**. Select **Help» NI-488.2 Help**.

Linux

Run GPIB Explorer by entering the following command:

`/usr/local/bin/gpibexplorer`

Select **Help»NI-488.2 Help**.

Conventions

The following conventions appear in this manual:

IEEE 488 and IEEE 488.2	*IEEE 488* and *IEEE 488.2* refer to the ANSI/IEEE Standard 488.1-2003 and the ANSI/IEEE Standard 488.2-1992, respectively, which define the GPIB.

Related Documentation

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

- ANSI/IEEE Standard 488.1-2003, *IEEE Standard Digital Interface for Programmable Instrumentation*
- ANSI/IEEE Standard 488.2-1992, *IEEE Standard Codes, Formats, Protocols, and Common Commands*
- *GPIB Hardware Installation Guide and Specifications*
- *NI-488.2 for Linux Installation Guide*
- *NI-488.2 for Mac OS X Getting Started*

1

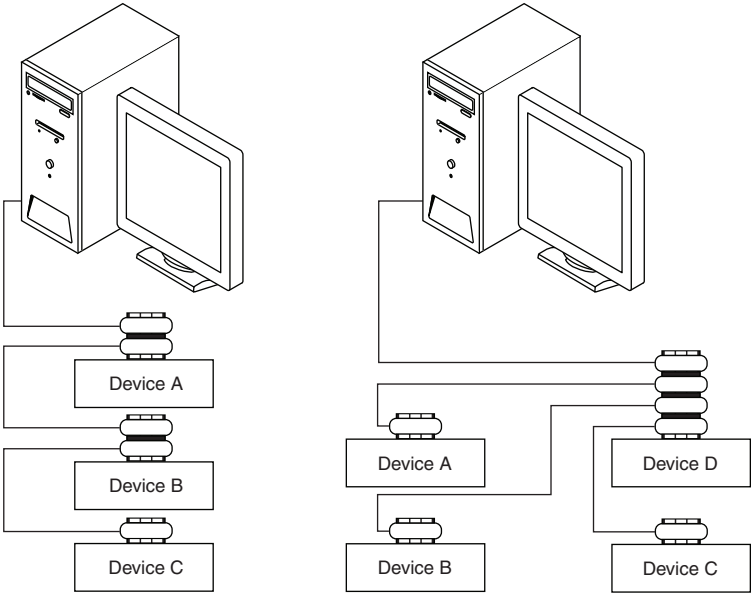
Introduction

This chapter describes how to set up your GPIB system.

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 configurations. Figure 1-1 shows the linear and star configurations.

Figure 1-1. Linear and Star System Configuration



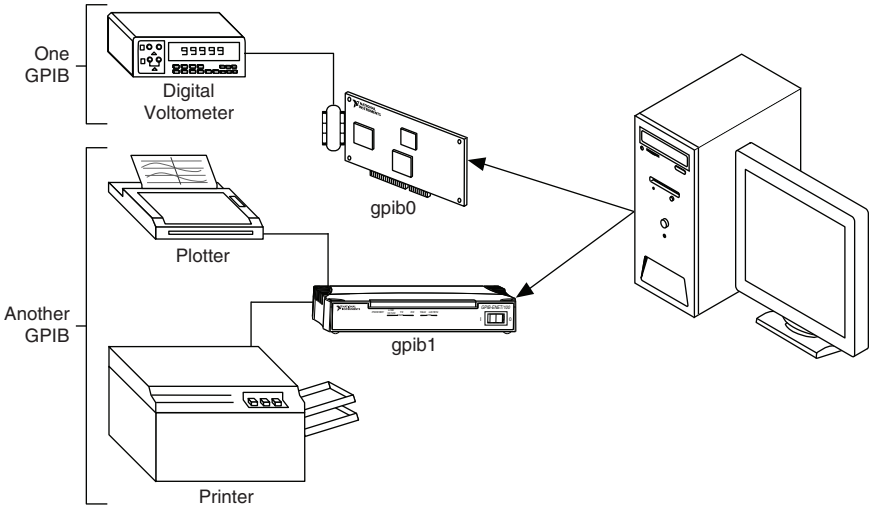
a. Linear Configuration

b. Star Configuration

Controlling More Than One Interface

Figure 1-2 shows an example of a multi-interface system configuration. `gpib0` is a PCI-GPIB and is the access interface for the voltmeter. `gpib1` is a GPIB-ENET/100 and is the access interface for the plotter and printer.

Figure 1-2. Example of Multiboard System Configuration



Configuration Requirements

To achieve the high data transfer rate that the GPIB was designed for, you must limit the number of devices on the bus and the physical distance between devices. The following restrictions are typical:

- A maximum separation of 4 m between any two devices and an average separation of 2 m over the entire bus.
- A maximum total cable length of 20 m.
- A maximum of 15 devices or controllers connected to each bus, with at least two-thirds powered on.

For high-speed (HS488) operation, the following restrictions apply:

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

If you want to exceed these limitations, you can use a bus extender to increase the cable length or a bus expander to increase the number of device loads. You can order bus extenders and expanders from National Instruments.

2

Measurement & Automation Explorer (Windows)

This chapter describes Measurement & Automation Explorer (MAX), an interactive utility you can use with the NI-488.2 software for Windows.

(OS X and Linux) NI-488.2 for OS X and NI-488.2 for Linux have a similar program called GPIB Explorer. For more information, refer to Chapter 3, *GPIB Explorer (OS X and Linux)*.

Overview

You can perform the following GPIB-related tasks in MAX:

- Establish basic communication with your GPIB instruments.
- Scan for instruments connected to your GPIB interface.
- Use Self-Test to troubleshoot NI-488.2 problems.
- Launch NI I/O Trace to monitor NI-488.2 or VISA API calls to GPIB interfaces.
- View information about your GPIB hardware and NI-488.2 software.
- Reconfigure GPIB interface settings.
- Locate additional help resources for GPIB and NI-488.2.

Starting Measurement & Automation Explorer

To start Measurement & Automation Explorer (MAX), select **Start»Programs»NI MAX**. **(Windows 8)** Click **NI Launcher** and select **NI MAX**.

The Measurement & Automation Explorer window shows the configuration tree at the left, the item view in the tabbed middle window, and a collapsible help pane at the right. Item view shows System Settings and System Resources when NI MAX starts.

Troubleshoot NI-488.2 Problems

Self-Test verifies that your GPIB hardware and the NI-488.2 software are installed correctly and able to perform basic I/O functions.

To start Self-Test, right-click the device name in the configuration tree, and select **Self-Test** from the shortcut menu.

Add a New GPIB Interface

For plug-and-play interfaces (such as PCI or USB), the system automatically detects and installs the hardware.

To add a new GPIB Ethernet interface to your system, complete the following steps:

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Right-click **Devices and Interfaces** and select **Create New**.
3. In the **Create New** dialog window, select **GPIB-ENET/100** or **GPIB-ENET/1000** and click **Finish**.

   The GPIB Ethernet Wizard appears.
4. Follow the prompts in the GPIB Ethernet Wizard to add your interface.
5. MAX automatically updates the list of installed GPIB interfaces. You also can select **View»Refresh** to update the list. Ethernet GPIB devices are listed under **Network Devices**.

Locate Your GPIB Interface

To locate a GPIB interface within MAX, complete the following steps:

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Expand **Devices and Interfaces**.
3. Depending on the GPIB interface, it is listed in its appropriate section:
   - Plug and Play GPIB interfaces, such as PCI-GPIB or GPIB-USB-HS, are listed directly under **Devices and Interfaces**.
   - GPIB interfaces installed in a PXI system are listed under that system.
   - GPIB Ethernet interfaces that have been added are listed under **Network Devices**.

Remove a GPIB Interface

To remove a Plug and Play interface from your computer, disconnect it, making sure to turn off the computer if the interface requires it.

To remove a GPIB Ethernet interface from your computer, you must manually delete it from within MAX by completing the following steps:

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Expand **Devices and Interfaces** and then expand **Network Devices**.
3. Right-click your GPIB Ethernet interface and select **Delete** from the context menu.
4. When prompted, confirm your selection.
5. Select **View»Refresh** to update the list of interfaces in Measurement & Automation Explorer.

Scan for GPIB Instruments

To scan for instruments connected to your GPIB interface or to add a new instrument to your system, complete the following steps:

1. Make sure that your instrument is powered on and connected to your GPIB interface.
2. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
3. Expand **Devices and Interfaces** and *Locate Your GPIB Interface*.
4. Right-click your GPIB interface and select **Scan for Instruments** from the drop-down menu that appears.

   Connected instruments appear beneath the GPIB interface in the Measurement & Automation Explorer configuration tree.

Instruments Not Found

If MAX reports that it did not find any instruments, make sure that your GPIB instruments are powered on and properly connected to the GPIB interface with a GPIB cable. Then, scan for instruments again, as described in the *Scan for GPIB Instruments* section.

Too Many Listeners on the GPIB

If MAX reports that it found too many Listeners on the GPIB, refer to the following possible solutions:

- If you have a running GPIB Analyzer with the GPIB handshake option enabled, disable the GPIB handshake option in the GPIB Analyzer.
- If you have a GPIB extender in your system, MAX cannot detect any instruments connected to your GPIB interface. Instead, you can verify communication with your instruments using the Interactive Control utility. To do so, select **Tools»NI-488.2»Interactive Control**. For more information about verifying instrument communication, type `help` at the Interactive Control command prompt.

Communicate with Your Instrument

To establish basic or advanced communication with your instruments, refer to the following sections.

For more information about instrument communication and a list of the commands that your instrument understands, refer to the documentation that came with your GPIB instrument. Most instruments respond to the `*IDN?` command by returning an identification string.

Basic Communication (Query/Write/Read)

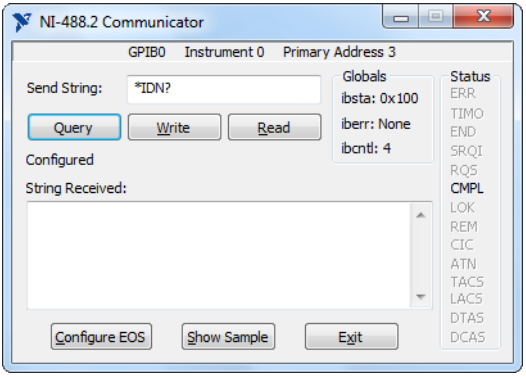
To establish basic communication with your instrument, use the NI-488.2 Communicator, as follows:

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Expand **Devices and Interfaces** and *Locate Your GPIB Interface*.
3. Select your GPIB interface.
4. If you have not already done so, scan for connected instruments. Right-click your GPIB interface and select **Scan for Instruments** from the drop-down menu that appears. Refer to the *Scan for GPIB Instruments* section for more information.

   MAX displays the connected instruments below your GPIB interface.
5. Right-click your GPIB instrument in the left window pane and select **Communicate with Instrument** from the drop-down menu that appears.

   The **NI-488.2 Communicator** dialog box appears, as shown in Figure 2-1.

Figure 2-1. NI-488.2 Communicator



6. Type a command in the **Send String** field and do one of the following:
   - To write a command to the instrument then read a response back, click the **Query** button.
   - To write a command to the instrument, click the **Write** button.
   - To read a response from the instrument, click the **Read** button.

To view sample C/C++ code that performs a simple query of a GPIB instrument, click the **Show Sample** button.

Advanced Communication

For advanced interactive communication with GPIB instruments, use the Interactive Control utility, as follows:

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Expand **Devices and Interfaces** and *Locate Your GPIB Interface*.
3. Right-click your GPIB interface and select **Interactive Control** from the drop-down menu that appears. Interactive Control automatically opens a session to the selected GPIB interface.
4. At the command prompt, type NI-488.2 API calls to communicate interactively with the your instrument. For example, you might use `ibdev`, `ibclr`, `ibwrt`, `ibrd`, and `ibonl`.

To view the help for Interactive Control, type `help` at the Interactive Control command prompt. For more information on using this utility, refer to Chapter 7, *Interactive Control Utility*.

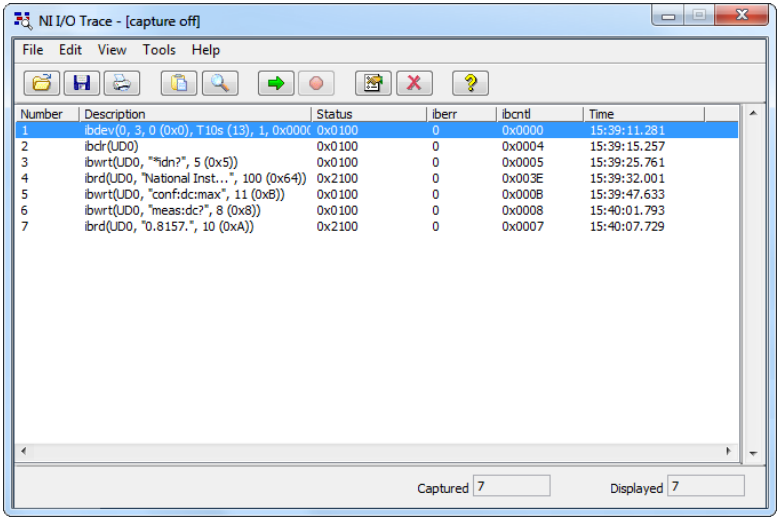
Monitor and Record NI-488.2 Calls

To monitor NI-488.2 calls, use NI I/O Trace, as follows:

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Select **Tools»NI I/O Trace** to open the application.
3. On the NI I/O Trace toolbar, click the green arrow button to start a capture.
4. Start the NI-488.2 application that you want to monitor.

   NI I/O Trace records and displays all NI-488.2 calls, as shown in Figure 2-2.

Figure 2-2. NI-488.2 Calls Recorded by NI I/O Trace



For more information about using NI I/O Trace, select **Help»Help Topics** in NI I/O Trace.

View or Change GPIB Interface Settings

To view or change interface settings for your GPIB instruments, complete the following steps:

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Expand **Devices and Interfaces** and *Locate Your GPIB Interface*.
3. Select your GPIB interface in the left window pane of MAX.
   The interface properties appear in the right window pane of MAX.
4. (Optional) Change the settings for your interface and click **Save** to apply the settings.

View GPIB Instrumentation Information

To view information about your GPIB instruments, complete the following steps:

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Expand **Devices and Interfaces** and *Locate Your GPIB Interface*.
3. Select your GPIB interface.
   MAX displays the connected instruments in the right window pane.
4. If you have not already done so, scan for connected instruments. Right-click your GPIB interface and select **Scan for Instruments** from the drop-down menu that appears. Refer to the *Scan for GPIB Instruments* section for more information.
5. Double-click the instrument displayed in the right window pane.
   MAX lists all the attributes for the instrument, such as the primary address, the secondary address (if applicable), the instrument's response to the identification query (`*IDN?`), and the GPIB interface number to which the device is connected.

Change GPIB Device Templates

For older NI-488.2 applications, you might need to modify one of the device templates to find a given GPIB instrument by name, for example, `ibfind("fluke45")`. Older applications still use `ibfind` instead of the preferred `ibdev` to obtain a device handle. In new applications, avoid using `ibfind` to obtain device handles and use `ibdev` instead. You can use `ibdev` to dynamically configure your GPIB device handle. `ibdev` also eliminates unnecessary device name requirements.

If you must modify a device template, run the GPIB Configuration utility.

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Select **Help»Help Topics»NI-488.2** to view the *NI-488.2 Help*.
3. Search for the topic named *How do I change a GPIB Device Template?* and click the link to open the GPIB Configuration utility.
4. Double-click the device template you want to modify, such as **DEV1**.
5. Rename the template as described in your application documentation.
6. Click the **OK** button twice to save your changes and exit.

Enable/Disable NI-488.2 DOS Support

NI-488.2 DOS support allows GPIB programs compiled for MS-DOS to run on 32-bit Windows. DOS support is enabled by default on 32-bit Windows systems.

Note NI-488.2 DOS support is available only on 32-bit Windows.

To disable NI-488.2 DOS support, complete the following steps:

1. Edit the config.nt file located in your Windows System32 directory (usually `c:\windows\system32`).
2. Add the `REM` prefix to the `Gpib-nt.com` line, as in the following example, to comment out the line.

   ```
   REM device=<path>\NI-488.2\DOSWIN16\Gpib-nt.com
   ```

   where *<path>* is the directory where you installed the NI-488.2 software.

To re-enable NI-488.2 DOS support:

1. Edit the config.nt file.
2. Remove the `REM` prefix from the `Gpib-nt.com` line.

Access Additional Help and Resources

To access additional help and resources for the NI-488.2 software and your GPIB hardware, refer to the following sections.

NI-488.2 Help

The *NI-488.2 Help* addresses questions you might have about NI-488.2 and includes a function reference and troubleshooting information. You can access the *NI-488.2 Help* as follows:

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Select **Help»Help Topics»NI-488.2**.

National Instruments GPIB Website

You can access the National Instruments GPIB website as follows:

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Select **Help»National Instruments on the Web»GPIB Home Page**.

View or Change GPIB-ENET/100 Network Settings

To view or change the network settings of your GPIB-ENET/100, refer to the following sections. For more information about your GPIB-ENET/100 network settings, refer to the GPIB-ENET/100 information in the *GPIB Hardware Guide*.

Device Configuration

Use the NI Ethernet Device Configuration utility if you need to manually configure the network parameters of the GPIB-ENET/100. If your network uses DHCP, the network configuration is performed automatically at startup and you do not need to run this utility unless you want to change the hostname. Consult your network administrator if you do not know whether your network uses DHCP.

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Expand **Devices and Interfaces** and then expand **Network Devices**.
3. Right-click your GPIB-ENET/100 interface and select **Device Configuration** from the drop-down menu that appears.

For more information about the NI Ethernet Device Configuration utility, refer to the GPIB-ENET/100 information in the *GPIB Hardware Installation Guide and Specifications*.

View or Change GPIB-ENET/1000 Network Settings

To view or change the network settings of your GPIB-ENET/1000, refer to the following sections. For more information about your GPIB-ENET/1000 network settings, refer to the GPIB-ENET/1000 information in the *GPIB Hardware Installation Guide and Specifications*.

Device Configuration

Use the GPIB Ethernet Device Configuration web page if you need to configure the GPIB-ENET/1000 network parameters manually. If your network uses DHCP, the network configuration is performed automatically at startup, and you do not need to run this utility unless you want to change the hostname. Consult your network administrator if you do not know whether your network uses DHCP.

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Expand **Devices and Interfaces** and then expand **Network Devices**.
3. Right-click your GPIB-ENET/1000 interface and select **Device Configuration** from the drop-down menu that appears. The GPIB Ethernet Device Configuration web page should launch in a browser window.

Update GPIB-ENET/1000 Firmware

You can run the Firmware Update utility as follows:

1. Start MAX as described in the *Starting Measurement & Automation Explorer* section.
2. Expand **Devices and Interfaces** and then expand **Network Devices**.
3. Right-click your GPIB-ENET/1000 interface and select **Device Configuration** from the drop-down menu that appears. The GPIB Ethernet Device Configuration web page should launch in a browser window.
4. In the **Details** section of the GPIB Ethernet Device Configuration web page, find the **Firmware** section and click **Update**.

For more information about the Firmware Update utility, refer to the GPIB-ENET/1000 information in the *GPIB Hardware Installation Guide and Specifications*.

3

GPIB Explorer (OS X and Linux)

This chapter describes GPIB Explorer, an interactive utility you can use with the NI-488.2 software for OS X and Linux.

You can perform the following GPIB-related tasks in GPIB Explorer:

- Add or remove GPIB interfaces.
- Reconfigure GPIB interface settings.
- Launch the NI-488.2 Troubleshooting Wizard to troubleshoot GPIB and NI-488.2 problems.
- Launch NI I/O Trace to monitor NI-488.2 calls to GPIB interfaces.
- Locate additional help resources for GPIB and NI-488.2.

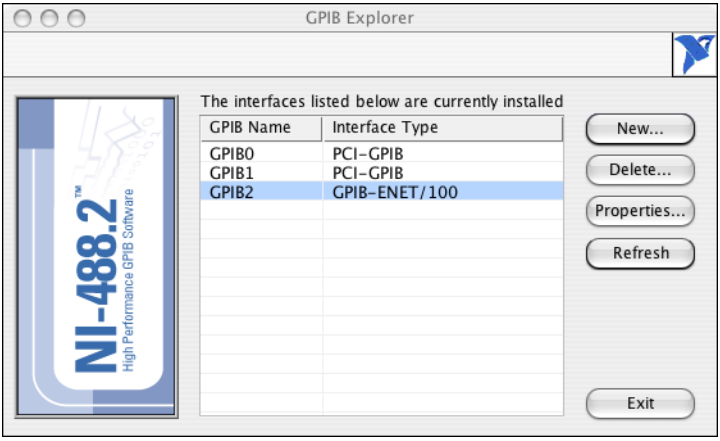
Starting GPIB Explorer

OS X

To start GPIB Explorer from the Finder, double-click **Applications»National Instruments» NI-488.2»GPIB Explorer**.

Figure 3-1 shows GPIB Explorer in OS X.

Figure 3-1. GPIB Explorer (OS X)



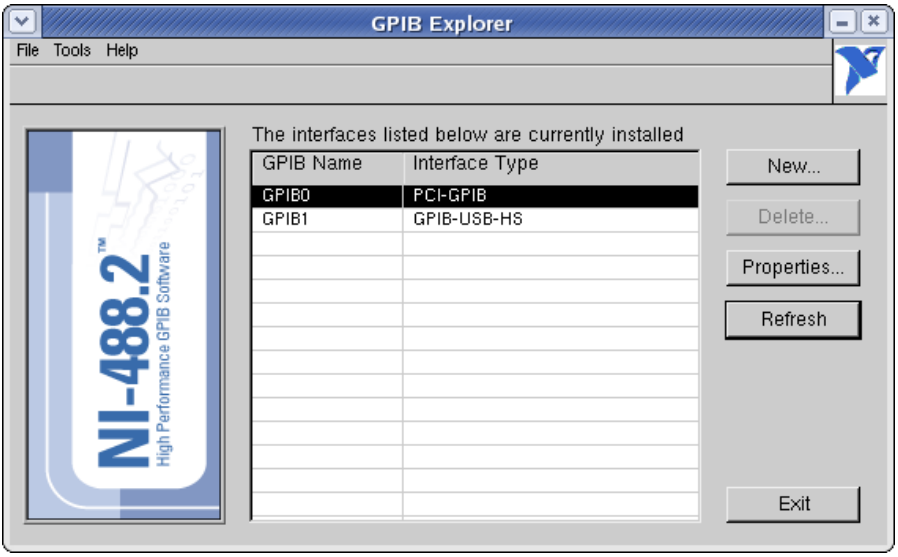
Linux

To start GPIB Explorer, enter the following command:

```
/usr/local/bin/gpibexplorer
```

Figure 3-2 shows GPIB Explorer in Linux.

Figure 3-2. GPIB Explorer (Linux)



Add a New GPIB Interface

To add a new GPIB interface to your system, complete the following steps:

Non-Plug and Play Interfaces (For Example, GPIB-ENET/100)

1. Start GPIB Explorer as described in the *Starting GPIB Explorer* section.
2. Click **New**.
3. Follow the prompts to add your GPIB interface to the system.

Plug and Play Interfaces (For Example, PCI-GPIB)

1. Close GPIB Explorer if it is running.
2. Physically add the interface into your system, making sure to shut down the system if your interface is not hot swappable. Refer to the *GPIB Hardware Installation Guide and Specifications* for more details on how to do this. To view the document, you need Acrobat Reader, which you can download from `www.adobe.com`.

   (OS X) The *GPIB Hardware Installation Guide and Specifications* is installed with NI-488.2. To access this document, double-click **Applications»National Instruments» NI-488.2»Documentation**.

   (Linux) The *GPIB Hardware Installation Guide and Specifications* is installed with NI-488.2. It is in the `/usr/local/natinst/ni4882/docs` directory.
3. Start GPIB Explorer as described in the *Starting GPIB Explorer* section. You should see your interface in the list of configured interfaces.

Delete a GPIB Interface

To remove a GPIB interface from your system, complete the following steps:

Non-Plug and Play Interfaces (For Example, GPIB-ENET/100)

1. Start GPIB Explorer as described in the *Starting GPIB Explorer* section.
2. Click your GPIB interface, and select **Delete**.
3. When prompted, click the **Yes** button to confirm the removal of your interface.

Plug and Play Interfaces (For Example, PCI-GPIB)

1. Close GPIB Explorer if it is running.
2. Physically remove the interface from your system, making sure to shut down the system if your interface is not hot swappable. Refer to the *GPIB Hardware Installation Guide and Specifications* for more details on how to do this. To view the document, you need Acrobat Reader, which you can download from `www.adobe.com`.

   (OS X) The *GPIB Hardware Installation Guide and Specifications* is installed with NI-488.2. To access this document, double-click **Applications»National Instruments» NI-488.2»Documentation**.

   (Linux) The *GPIB Hardware Installation Guide and Specifications* is installed with NI-488.2. It is in the `/usr/local/natinst/ni4882/docs` directory.
3. Start GPIB Explorer as described in the *Starting GPIB Explorer* section. The interface you just removed should not be in the list of configured interfaces.

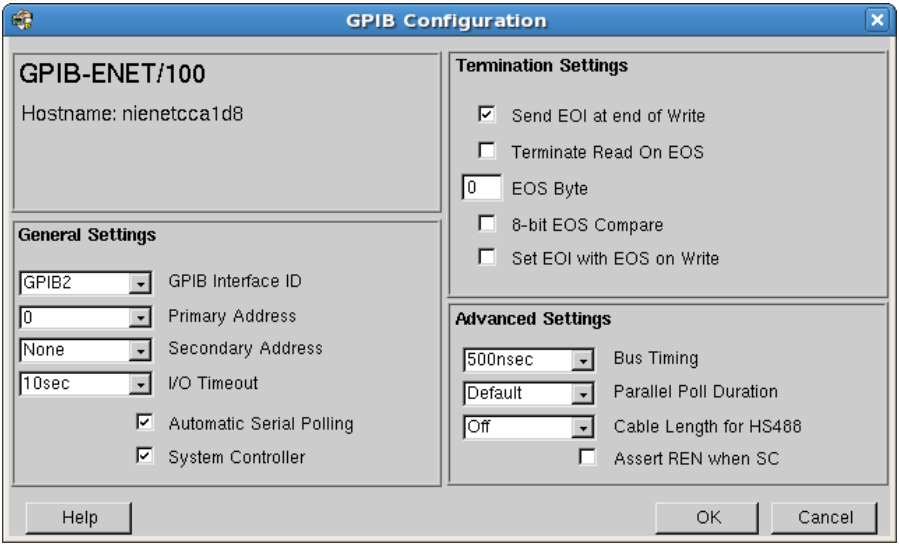
View or Change GPIB Interface Settings

To view or change your interface settings, complete the following steps:

1. Start GPIB Explorer as described in the *Starting GPIB Explorer* section.
2. Click your GPIB interface, and click **Properties**.

   The **Properties** dialog box appears.

Figure 3-3. GPIB Configuration



3. (Optional) Change the settings for your interface, then click the **OK** button.

Access Additional Help and Resources

To access additional help and resources for the NI-488.2 software and your GPIB hardware, refer to the following sections.

NI-488.2 Help

The *NI-488.2 Help* addresses questions you might have about NI-488.2 and includes a function reference and troubleshooting information. You can access the *NI-488.2 Help* as follows:

1. Start GPIB Explorer as described in the *Starting GPIB Explorer* section.
2. Select **Help»NI-488.2 Help** from the menu bar.

National Instruments GPIB Website

1. Start GPIB Explorer as described in the *Starting GPIB Explorer* section.
2. Select **Help»NI GPIB Home Page** from the menu bar to access the National Instruments website for GPIB.

View or Change GPIB Ethernet Device Network Settings

To view or change the network settings of your GPIB Ethernet device, refer to the following sections. For more information about your network settings, refer to the *GPIB Hardware Guide*. To view the *GPIB Hardware Installation Guide and Specifications*, you need Adobe Acrobat Reader, which you can download from `www.adobe.com`.

(**OS X**) The *GPIB Hardware Installation Guide and Specifications* is installed with NI-488.2. To access this document, double-click **Applications»National Instruments»NI-488.2» Documentation**.

(**Linux**) The *GPIB Hardware Installation Guide and Specifications* is installed with NI-488.2. It is in the `/usr/local/natinst/ni4882/docs` directory.

4

Developing Your NI-488.2 Application

This chapter describes how to develop an NI-488.2 application using the NI-488.2 API.

Interactive Instrument Control

Before you write your NI-488.2 application, you might want to use the Interactive Control utility to communicate with your instruments interactively by typing individual commands rather than issuing them from an application. You can also use the Interactive Control utility to learn to communicate with your instruments using the NI-488.2 API. For specific device communication instructions, refer to the documentation that came with your instrument. For information about using the Interactive Control utility and detailed examples, refer to Chapter 7, *Interactive Control Utility*. To view the help for Interactive Control, type `help` at the Interactive Control command prompt.

Windows

1. Start MAX as described in Chapter 2, *Measurement & Automation Explorer (Windows)*.
2. Select **Tools»NI-488.2»Interactive Control**.
3. At the command prompt, type NI-488.2 API calls to communicate interactively with your instrument. For example, you might use `ibdev`, `ibclr`, `ibwrt`, `ibrd`, and `ibonl`.

OS X

1. Double-click **Applications»National Instruments»NI-488.2»Interactive Control**.
2. At the command prompt, type NI-488.2 API calls to communicate interactively with your instrument. For example, you might use `ibdev`, `ibclr`, `ibwrt`, `ibrd`, and `ibonl`.

Linux

1. To launch the Interactive Control utility, enter the following command:

   `/usr/local/bin/gpibintctrl`
2. At the command prompt, type NI-488.2 API calls to communicate interactively with your instrument. For example, you might use `ibdev`, `ibclr`, `ibwrt`, `ibrd`, and `ibonl`.

Choosing Your Programming Methodology

Based on your development environment, you can select a method for accessing the driver, and based on your NI-488.2 programming needs, you can choose how to use the NI-488.2 API.

Choosing a Method to Access the NI-488.2 Driver

Windows

Applications using the older GPIB32 API can access the NI-488.2 dynamic link library (DLL), `gpib-32.dll`, either by using an NI-488.2 application interface or by direct access.

Applications using the new NI4882 API can access the NI-488.2 dynamic link library (DLL), `ni4882.dll`, by using an NI-488.2 application interface or by direct access.

NI-488.2 Application Interfaces

You can use an application interface if your program is written in Microsoft Visual C/C++ (6.0 or later), Borland C/C++ (5.02 or later), Microsoft Visual Basic (6.0), or any .NET programming language. Otherwise, you must access the dynamic link library directly.

For more information about application interfaces, refer to *NI-488.2 Application Interface Files* in the *NI-488.2 Help*.

Direct Entry Access

You can access the DLL directly from any programming environment that allows you to request addresses of variables and calls that a DLL exports. The dynamic link libraries export pointers to each of the global status functions or variables and all the NI-488.2 calls.

For more information about direct entry access, refer to *Directly Accessing the ni4882.dll Exports in C* or *Directly Accessing the gpib-32.dll Exports in C* in the *NI-488.2 Help*.

OS X

NI-488.2 provides `NI4882.framework`, which can be used from 32-bit and 64-bit C/C++ applications. Refer to the *Language-Specific Programming Instructions for OS X* section for more details on how to develop your application.

Linux

NI-488.2 provides the `libni4882.so` dynamically-linked library which can be used from 32-bit and 64-bit C/C++ applications. Refer to the *Language-Specific Programming Instructions for Linux* section for more details on how to develop your application.

Differences Between the GPIB32 API and NI4882 API

The NI-488.2 for Windows 2.6, NI-488.2 for Linux 3.2, and NI-488.2 for Mac OS X 3.2 releases officially add support for a new API as part of the 64-bit application interface. Every effort has been made to have the new NI4882 API closely match the existing GPIB32 API while incorporating API design best practices. To use the new API, you must recompile applications using the new header and object files. The following list describes the major changes in the NI4882 API:

- Judicious application of the `const` keyword has been added where appropriate.
- Wide variants of functions now use the `wchar_t` instead of `unsigned short` type.
- Functions taking in parameters that describe a pointer length now use `size_t` types.
- Status variables now use the `unsigned int` type.
- `ThreadIbcntl` has been removed. Macros redirect calls to `ThreadIbcnt`.
- Global status functions have been added. These are `Ibsta`, `Iberr`, and `Ibcnt`. New code should use these functions instead of `ibsta`, `iberr`, or `ibcnt`/`ibcntl`.
- Long-term deprecated functions have been completely removed.
- Most functions with an `ibconfig` have been removed. Using `ibconfig` is recommended for new code. Existing functions redirect to using `ibconfig` using macros. These are the affected functions:
  - `ibpad`
  - `ibsad`
  - `ibtmo`
  - `ibeot`
  - `ibrsc`
  - `ibsre`
  - `ibeos`
  - `ibdma`
  - `ibist`
  - `ibrsv`
- Many macro definitions have been improved for programmatic safety.

  Modifying existing applications to use the NI4882 API should require minimal changes. In most cases, using the new include file (`ni4882.h` instead of `ni488.h`) and linking to the new object file:

  (Windows) `ni4882.obj` instead of `gpib-32.obj`

  (Linux) `libni4882.so` instead of `libgpibapi.so`

  (OS X) `ni4882.framework` instead of `ni488.framework`

  There may still be warnings due to changes to the signed property of the status variable type.

Complications may arise in several uncommon use cases. The following issues have been encountered:

- Storing function pointers for the `ibnotify` callback. This causes a type mismatch on the assignment. To solve this, fix the function prototype of the callback to use `unsigned int` for the status parameters.
- Using function pointers to `ibfind`. This causes a preprocessor error because the `ibfind` macro requires a one-parameter argument. To solve this, point to `ibfindA` or `ibfindW`, depending on the unicode convention in your application.
- Configuration functions show up in NI I/O Trace as `ibconfig` calls. This is because macros redirect those calls to use `ibconfig`. Avoid confusion by using `ibconfig` directly.

In most cases, applications written in the NI4882 API will continue to work on older versions of the NI-488.2 for Windows software, back to version 1.7. Certain new `ibask` and `ibconfig` options break this backwards compatibility, and those options are easily avoidable by using alternative options. Existing applications using the GPIB32 API continue to execute unchanged. The GPIB32 API will continue to exist, but is available only for 32-bit applications. Applications written in the NI4882 API compile on both 32-bit and 64-bit environments. To port an application to a 64-bit environment requires that the application migrate to the NI4882 API and be recompiled.

The following NI4882 API constructs break API compatibility with older versions of NI-488.2:

- `ibask` (`IbaEOS`)
- `ibconfig` (`IbcEOS`)

Choosing How to Use the NI-488.2 API

The NI-488.2 API has two subsets of calls to meet your application needs. Both of these sets, the traditional calls and the multi-device calls, 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 traditional NI-488.2 calls are sufficient. If you have a complex configuration with one or more interfaces and multiple devices, use the multi-device NI-488.2 calls. Whichever option you choose, bus management operations necessary for device communication are performed automatically.

The following sections describe some differences between the traditional NI-488.2 calls and the multi-device NI-488.2 calls.

Communicating with a Single GPIB Device

If your system has only one device attached to each interface, the traditional NI-488.2 calls are probably sufficient for your programming needs. A typical NI-488.2 application with a single device has three phases:

- Initialization: use `ibdev` to get a handle and use `ibclr` to clear the device.
- Device Communication: use `ibwrt`, `ibrd`, `ibtrg`, `ibrsp`, and `ibwait` to communicate with the device.
- Cleanup: use `ibonl` to put the handle offline.

Refer to the sample applications that are installed with the NI-488.2 software to see detailed examples for different GPIB device types.

For NI-488.2 applications that need to control the GPIB in non-typical ways—for example, to communicate with non-compliant GPIB devices—there is a set of low-level functions that perform rudimentary GPIB functions. If you use these functions, you need to understand GPIB management details such as how to address talkers and listeners. Refer to Appendix A, *GPIB Basics*, for some details on GPIB management.

The set of low-level functions are called board-level functions. They access the interface directly and require you to handle the addressing and bus management protocol. These 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 interface configurations.
- Managing the bus in non-typical ways.

Board-level functions that an NI-488.2 application might use include the following—`ibcmd`, `ibrd`, `ibwrt`, and `ibconfig`. For a detailed list, refer to the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

Using Multiple Interfaces and/or Multiple Devices

When your system includes an interface that must access multiple devices, use the multi-device NI-488.2 calls, which can perform the following tasks with a single call:

- Find the Listeners on the bus using `FindLstn`.
- Find a device requesting service using `FindRQS`.
- Determine the state of the SRQ line, or wait for SRQ to be asserted using `TestSRQ` or `WaitSRQ`.
- Address multiple devices to receive a command using `SendList`.

You can mix board-level traditional NI-488.2 calls with the multi-device NI-488.2 calls to have access to all the NI-488.2 functionality.

Checking Status with Global Functions

For applications accessing the NI4882 API, each NI-488.2 call updates three global functions to reflect the status of the device or board you are using. These global status functions are the status word (`Ibsta`), the error function (`Iberr`), and the count function (`Ibcnt`). They contain useful information about the performance of your application. Your application should check these functions after each NI-488.2 call. For more information about each status function, refer to the following sections.

For applications accessing the older GPIB32 API (including the Visual Basic 6.0 application interface), use the equivalent global variables. These global status variables are the status word (`ibsta`), the error variable (`iberr`), and the count variables (`ibcnt` and `ibcntl`). `ibcnt` is defined to be the type `int`, while `ibcntl` is the size of type long `int`. For all cases, if the sizes of `ibcnt` and `ibcntl` are the same, `ibcnt` and `ibcntl` are equal. For cross-platform compatibility, all applications should use `ibcntl`.

For applications accessing the newer NI4882 API, use the global function calls rather than the global variables. The global functions replace the global variables with the newer NI4882 API.

Note If your application is a multithreaded application, refer to the *Writing Multithreaded NI-488.2 Applications* section of Chapter 8, *NI-488.2 Programming Techniques*.

Status Word (Ibsta)

All calls update a global status function, `Ibsta`, which contains information about the state of the GPIB and your GPIB hardware. 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. When using the GPIB32 API, `ibsta` is the global variable.

Each bit in `Ibsta` can be set for device-level traditional NI-488.2 calls (dev), board-level traditional NI-488.2 calls and multi-device NI-488.2 calls (brd), or all (dev, brd). `Ibsta` is a 32-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.

Table 4-1 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 status condition, refer to Appendix B, *Status Word Conditions*.

Table 4-1. Status Word Layout

Mnemonic	Bit Pos	Hex Value	Type	Description
ERR	15	8000	dev, brd	NI-488.2 error
TIMO	14	4000	dev, brd	Time limit exceeded

Table 4-1. Status Word Layout (Continued)

Mnemonic	Bit Pos	Hex Value	Type	Description
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 defines each `Ibsta` status bit. 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 an NI-488.2 error, use the following statement after each NI-488.2 call:

```
if (Ibsta() & ERR)
    printf("NI-488.2 error encountered");
```

Error Function (Iberr)

If the ERR bit is set in `Ibsta`, an NI-488.2 error has occurred. When an error occurs, the error type is specified by `Iberr`. To check for an NI-488.2 error, use the following statement after each NI-488.2 call:

```
if (Ibsta() & ERR)
    printf("NI-488.2 error %d encountered", Iberr());
```

Note The value in `Iberr()` is meaningful as an error only when the ERR bit is set in `Ibsta`, indicating that an error has occurred.

For more information about error codes and solutions, refer to Chapter 5, *Debugging Your Application*, or Appendix C, *Error Codes and Solutions*.

Count Function (Ibcnt)

The count function is updated after each read, write, or command function. In addition, `Ibcnt` is updated after specific 488.2-style functions in certain error cases. Refer to the *NI-488.2 Help* function reference for an explanation of how each function uses the count function.

`Ibcnt` is defined to be the type `unsigned int`.

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.

Using Interactive Control to Communicate with Devices

Before you begin writing your application, you might want to use the Interactive Control utility to communicate with your instruments interactively by typing in commands from the keyboard rather than from an application. You can use the Interactive Control utility to learn to communicate with your instruments using the NI-488.2 API. For specific device communication instructions, refer to the user manual that came with your instrument. For information about using the Interactive Control utility and detailed examples, refer to Chapter 7, *Interactive Control Utility*.

Programming Models

Applications That Communicate with a Single GPIB Device

This section describes items you should include in your application and provides general program steps with an NI-488.2 example.

Items to Include

Include the following items in your application:

- Header files—In a C application, include the header file `ni4882.h`, which contains prototypes for the NI-488.2 calls and constants that you can use in your application.
- Error checking—Check for errors after each NI-488.2 call.
- Error handling—Declare and define a function to handle NI-488.2 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("NI-488.2 error");
  }
  ```

General Program Steps and Examples

The following steps show you how to use the traditional NI-488.2 device-level calls in your application. The NI-488.2 software includes the `devquery` source code example to demonstrate these principles.

Initialization

Step 1. Open a Device

Use `ibdev` to open a device handle. The `ibdev` function requires the following parameters:

- Connect board index (typically 0, for `GPIB0`).
- Primary address for the GPIB instrument (refer to the instrument user manual or use the `FindLstn` function to dynamically determine the GPIB address of your GPIB device, as described in *Step 2. Determine the GPIB Address of Your Device* in the *Applications That Use Multiple Interfaces or Communicate with Multiple GPIB Devices* section).
- 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 set to 1 so that EOI is asserted with the last byte of writes).
- EOS detection mode (typically 0 if the GPIB instrument does not use EOS characters).

A successful `ibdev` call returns a device handle, `ud`, that is used for all device-level traditional NI-488.2 calls that communicate with the GPIB instrument.

Step 2. Clear the Device

Use `ibclr` to clear the device. This resets the internal functions of the device to the default state.

Device Communication

Step 3. Communicate with the Device

Communicate with the device by sending it the `"*IDN?"` query and then reading back the response. Many devices respond to this query by returning a description of the device. Refer to the documentation that came with your GPIB device to see specific instructions on the proper way to communicate with it.

Step 3a.

Use `ibwrt` to send the `"*IDN?"` query command to the device.

Step 3b.

Use `ibrd` to read the response from the device.

Continue communicating with the GPIB device until you are finished.

Cleanup

Step 4. Place the Device Offline before Exiting Your Application

Use `ibonl` to put the device handle offline before you exit the application.

Applications That Use Multiple Interfaces or Communicate with Multiple GPIB Devices

This section describes items you should include in your application and provides general program steps with an NI-488.2 example.

Items to Include

Include the following items in your application:

- Header files—In a C application, include the header file `ni4882.h`, which contains prototypes for the NI-488.2 calls and constants that you can use in your application.
- Error checking—Check for errors after each NI-488.2 call.
- Error handling—Declare and define a function to handle NI-488.2 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("NI-488.2 error");
  }
  ```

General Program Steps and Examples

The following steps show you how to use the multi-device NI-488.2 calls in your application. The NI-488.2 software includes the `4882query` source code examples to demonstrate these principles.

Initialization

Step 1. Become Controller-In-Charge (CIC)

Use `SendIFC` to initialize the bus and the GPIB interface so that the GPIB interface is Controller-In-Charge (CIC). The only argument of `SendIFC` is the GPIB interface number, typically 0 for `GPIB0`.

Step 2. Determine the GPIB Address of Your Device

Use `FindLstn` to find all the devices attached to the GPIB. The `FindLstn` function requires the following parameters:

- Interface number (typically 0, for `GPIB0`).
- A list of primary addresses, terminated with the `NOADDR` constant.
- A list for reported GPIB addresses of devices found listening on the GPIB.
- Limit, which is the number of the GPIB addresses to report.

Use `FindLstn` to test 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 is stored in the GPIB addresses list. When you have the list of GPIB addresses, you can determine which one corresponds to your instrument and use it for subsequent calls.

Alternately, 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 16-bit value is 0x01; otherwise, if the primary address is 1 and the secondary address is 0x67, then the 16-bit value is 0x6701.

Step 3. Initialize the Devices

Use `DevClearList` to clear the devices on the GPIB. The first argument is the GPIB interface number. The second argument is the list of GPIB addresses that were found to be listening as determined in Step 2.

Device Communication

Step 4. Communicate with the Devices

Communicate with the devices by sending them the `"*IDN?"` query and then reading back the responses. Many devices respond to this query by returning a description of the device. Refer to the documentation that came with your GPIB devices to see specific instruction on the proper way to communicate with them.

Step 4a.

Use `SendList` to send the `"*IDN?"` query command to multiple GPIB devices. The address is the list of GPIB devices to be queried. The buffer that you pass to `SendList` is the command message to the device.

Step 4b.

Use `Receive` for each device to read the responses from each device.

Continue communicating with the GPIB devices until you are finished.

Cleanup

Step 5. Place the Interface Offline before Exiting Your Application

Use `ibonl` to put the interface offline before you exit the application.

Language-Specific Programming Instructions for Windows

The following sections describe how to develop, compile, and link your Windows NI-488.2 applications using various programming languages.

Microsoft Visual C/C++ (Version 6.0 or Later)

Before you compile your application, include the following line at the beginning of your program:

```
#include "ni4882.h"
```

The "NIEXTCCOMPILERSUPP" environment variable is provided as an alias to the location of C language support files. You can use this variable when compiling and linking an application.

With Microsoft Visual C++ (Version 6.0 or later), to compile and link a Win32 console application named cprog in a DOS shell or Visual C++'s Command Prompt using the environment variable, "NIEXTCCOMPILERSUPP", type in the following on the command line:

```
cl /I"%NIEXTCCOMPILERSUPP%\include" cprog.c
"%NIEXTCCOMPILERSUPP%\lib32\msvc\ni4882.obj" /MD
```

With Microsoft Visual C++ (Version 8.0 or later), to compile and link a Win64 console application named cprog in the Visual C++ x64 Command Prompt using the environment variable, "NIEXTCCOMPILERSUPP", type in the following on the command line:

```
cl /I"%NIEXTCCOMPILERSUPP%\include" cprog.c
"%NIEXTCCOMPILERSUPP%\lib64\msvc\ni4882.obj" /MD
```

Borland C/C++ (Version 5.0.2 or Later)

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

```
#include "ni4882.h"
```

The "NIEXTCCOMPILERSUPP" environment variable is provided as an alias to the location of C language support files. You can use this variable when compiling and linking an application.

To compile and link a Win32 console application named cprog in a DOS shell using the environment variable, "NIEXTCCOMPILERSUPP", type in the following on the command line:

```
bcc32 -I"%NIEXTCCOMPILERSUPP%\include" -w32 cprog.c
"%NIEXTCCOMPILERSUPP%\lib32\borland\ni4882.obj"
```

Borland/CodeGear/Embarcadero does not have a 64-bit compiler at the time of this writing.

Visual Basic (Version 6.0)

With Visual Basic, you can access the traditional NI-488.2 calls as subroutines, using the BASIC keyword `CALL` followed by the traditional NI-488.2 call name, or you can access them using the `il` set of functions. With some of the NI-488.2 calls (for example `ibrd` and `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. For more information about function syntax for Visual Basic, refer to the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

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

Direct Entry with C

Direct entry is available for 32-bit and 64-bit `ni4882.dll`s and the 32-bit `gpib-32.dll`.

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

DLL Exports

`gpib-32.dll` exports pointers to the global variables and all of the NI-488.2 calls. 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 that have string parameters such as `ibfind`, `ibrdf`, and `ibwrtf`, all the NI-488.2 call names are exported from the DLL. Thus, to use direct entry to access a particular function and to get a pointer to the exported function, you just need to call `GetProcAddress` passing the name of the function as a parameter. For more information about the parameters to use when you invoke the function, refer to the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

The functions such as `ibfind`, `ibrdf`, and `ibwrtf` all require an argument that is a name. `ibfind` requires an interface or device name and `ibrdf` and `ibwrtf` require a file name. Because Windows supports both ASCII (8-bit) and Unicode (16-bit) characters, the DLLs export both ASCII and Unicode versions of these functions. The ASCII versions are named `ibfindA`, `ibrdfA`, and `ibwrtfA`. The Unicode versions are named `ibfindW`, `ibrdfW`, and `ibwrtfW`. You can use either the Unicode or ASCII versions of these functions with Windows.

In addition to pointers to the status functions or variables and a handle to the loaded DLL, you must define the direct entry prototypes for the functions you use in your application. For the prototypes for each function exported by the DLL, refer to the appropriate header file—`ni4882.h` for ni4882 format or `ni488.h` for gpib-32 format. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

For more information about direct entry, refer to the help for your development environment.

Directly Accessing the ni4882.dll Exports

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

```
#include <windows.h>
#include "ni4882.h"
```

In your Windows application, you first need to load `ni4882.dll`. The following code fragment shows you how to call the `LoadLibrary` function and check for an error:

```
HINSTANCE ni4882Lib = NULL;
ni4882Lib=LoadLibrary("NI4882.DLL");
if (ni4882Lib == NULL) {
    return FALSE;
}
```

For the prototypes for each function, refer to the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

For functions that return an integer value, like `ibdev`, you need to cast the pointer as:

```
int (_stdcall *Pname)
```

where `*Pname` is the name of the pointer to the function. For calls that return an unsigned int value, such as `ibwrt`, you need to cast the pointer to the function as:

```
unsigned int (_stdcall *Pname)
```

where `*Pname` is the name of the pointer to the function. For functions that do not return a value, like `FindLstn` or `SendList`, you need to cast the pointer as:

```
void (_stdcall *Pname)
```

where `*Pname` is the name of the pointer to the function. It is followed by the function's list of parameters as described in the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

An example of how to declare the function pointer and parameter list for `ibdev` and `ibonl` follows:

```
int (_stdcall *Pibdev)(int ud, int pad, int sad, int tmo, int eot,
int eos);
unsigned int (_stdcall *Pibonl)(int ud, int v);
```

Your Windows application needs to use `GetProcAddress` to get the addresses of the function your application needs. The following code fragment shows you how to get the addresses of the pointers to the thread-specific status functions and any calls your application needs:

```
/* Pointers to NI-488.2 thread-specific status functions */
static unsigned int (__stdcall *PThreadIbsta)(void);
static unsigned int (__stdcall *PThreadIberr)(void);
static unsigned int (__stdcall *PThreadIbcnt)(void);
static int(__stdcall *Pibdev) (int ud, int pad, int sad, int tmo, int
eot, int eos);
static unsigned int(__stdcall *Pibonl)(int ud, int v);
PThreadIbsta = (unsigned int (__stdcall *)(void))
GetProcAddress(ni4882Lib, "ThreadIbsta");
PThreadIberr = (unsigned int (__stdcall *)(void))
GetProcAddress(ni4882Lib, "ThreadIberr");
PThreadIbcnt = (unsigned int (__stdcall *)(void))
GetProcAddress(ni4882Lib, "ThreadIbcnt");
Pibdev = (int (__stdcall *) (int, int, int, int, int,
int))GetProcAddress(ni4882Lib, "ibdev");
Pibonl = (unsigned int (__stdcall *)(int,
int))GetProcAddress(ni4882Lib, "ibonl");
```

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

```
if ((NULL == PThreadIbsta) ||
    (NULL == PThreadIberr) ||
    (NULL == PTheadIbcnt) ||
    (NULL == Pibdev) ||
    (NULL == Pibonl)) {
/* Free the ni4882 library */
    FreeLibrary(ni4882Lib);
    printf("GetProcAddress failed.");
}
```

Your Windows application must de-reference the pointer to access the function call. The following code shows you how to call a function and access the status function from within your application:

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

Before exiting your application, you need to free `ni4882.dll` with the following command:

```
FreeLibrary(ni4882Lib);
```

For more examples of directly accessing `ni4882.dll`, refer to the direct entry sample programs `dlldevquery.c` and `dll4882query.c`, installed with the NI-488.2 software. For more information about direct entry, refer to the help for your development environment.

Directly Accessing the gpib-32.dll Exports

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

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

#include <windows.h>
#include "ni488.h"

#ifdef __cplusplus
}
#endif
```

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

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

For the prototypes for each function, refer to the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

For functions that return an integer value, like `ibdev` or `ibwrt`, the pointer to the function needs to be cast as follows:

```
int (_stdcall *Pname)
```

where `*Pname` is the name of the pointer to the function. For functions that do not return a value, like `FindLstn` or `SendList`, the pointer to the function needs to be cast as follows:

```
void (_stdcall *Pname)
```

where `*Pname` is the name of the pointer to the function. They are followed by the function's list of parameters as described in the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

Following is an example of how to cast the function pointer and how the parameter list is set up for `ibdev` and `ibonl` functions:

```
int (_stdcall *Pibdev)(int ud, int pad, int sad, int tmo, int eot,
int eos);
int (_stdcall *Pibonl)(int ud, int v);
```

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

```
/* 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 shows you 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 shows you how to call a function and access the status variable from within your application:

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

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

```
FreeLibrary(Gpib32Lib);
```

For more information about direct entry, refer to the help for your development environment.

Language-Specific Programming Instructions for OS X

The following information describes how to develop, compile, and link your OS X NI-488.2 applications.

Before you compile your application, remember to include the following line at the beginning of your program:

```
#include <NI4882/ni4882.h>
```

To compile and link your application, include `NI4882.framework` in your project. The framework is located at `/Library/Frameworks`.

To compile and link your application in a Terminal Shell, type the following code on the command line:

```
cc cprog.c -framework NI4882
```

or

```
gcc cprog.c -framework NI4882
```

To build a 32-bit application on a 64-bit system, you must provide the -m32 option as shown in the following examples:

```
cc -m32 cprog.c -framework NI4882
```

or

```
gcc -m32 cprog.c -framework NI4882
```

Language-Specific Programming Instructions for Linux

The following information describes how to develop, compile, and link your Linux NI-488.2 applications for the NI4882 API.

Before you compile your application, remember to include the following line at the beginning of your program:

```
#include <ni4882.h>
```

Your application must link with the NI-488.2 dynamic library `libni4882.so`. There are two ways to load a dynamic library on Linux—*static* and *dynamic*.

To have the library *statically* loaded at the time your application starts, compile and link your application as shown in the following examples:

```
gcc prog.c -lni4882
```

or

```
g++ prog.cpp -lni4882
```

To build a 32-bit application on a 64-bit system, you must provide the -m32 option as shown in the following examples:

```
gcc -m32 prog.c -lni4882
```

or

```
g++ -m32 prog.cpp -lni4882
```

To have the library *dynamically* loaded on demand when your application accesses the library, include `ni4882.o` during the link phase of your application, as shown in the following examples:

```
gcc prog.c ni4882.o -ldl
```

or

```
g++ prog.cpp ni4882.o -ldl
```

The file `ni4882.o` contains code to dynamically load the library. This file is located in `/usr/local/natinst/ni4882/lib` and `/usr/local/natinst/ni4882/lib64`.

The advantage of the latter way of compiling and linking your application is that it allows your application to run regardless of whether the NI-488.2 software is installed, as long as it does not make any NI-488.2 calls.

5

Debugging Your Application

This chapter describes several ways to debug your application.

NI I/O Trace

The NI I/O Trace utility monitors NI-488.2 API calls made by NI-488.2 applications. If an application does not have built-in error detection handling, you can use NI I/O Trace to determine which NI-488.2 call is failing.

Starting NI I/O Trace

Windows

To start NI I/O Trace, complete the following steps:

1. Start MAX as described in Chapter 2, *Measurement & Automation Explorer (Windows)*.
2. From the menu bar, select **Tools»NI I/O Trace**.

OS X and Linux

To start NI I/O Trace, complete the following steps:

1. Start GPIB Explorer as described in Chapter 3, *GPIB Explorer (OS X and Linux)*.
2. From the menu bar, select **Tools»NI I/O Trace**.

Debugging Existing Applications

Once you know which NI-488.2 call fails, refer to Appendix B, *Status Word Conditions*, and Appendix C, *Error Codes and Solutions*, for help understanding why the NI-488.2 call failed. This information is also available in the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

Performance Considerations

NI I/O Trace can slow down the performance of your NI-488.2 application, and certain configurations of NI I/O Trace have a larger impact on performance than others. For example, configuring NI I/O Trace 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 your system. For this reason, use NI I/O Trace only while you are debugging your application or in situations where performance is not critical.

For more information about using NI I/O Trace, select **Help»Help Topics** in NI I/O Trace.

Global Status Functions

At the end of each NI-488.2 call, the global status functions (`Ibsta`, `Iberr`, and `Ibcnt`) are updated. If you are developing an NI-488.2 application, you should check for errors after each NI-488.2 call. If a NI-488.2 call failed, the high bit of `Ibsta` (the `ERR` bit) is set. For a failed NI-488.2 call, `Iberr` contains a value that defines the error. In some error cases, the value in `Ibcnt` contains even more error information.

Once you know which NI-488.2 call fails, refer to Appendix B, *Status Word Conditions*, and Appendix C, *Error Codes and Solutions*, for help understanding why the NI-488.2 call failed. This information is also available in the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

NI-488.2 Error Codes

The error function, `Iberr`, is meaningful only when the `ERR` bit in the status function, `Ibsta`, is set. For a detailed description of each error and possible solutions, refer to Appendix C, *Error Codes and Solutions*.

Configuration Errors

Several applications require customized configuration of the NI-488.2 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 reconfigure from your application using the `ibconfig` function or reconfigure using the GPIB Configuration utility.

Note National Instruments recommends using `ibconfig` to modify the configuration.

If your application uses `ibconfig`, it works properly regardless of the previous configuration. For more information about using `ibconfig`, refer to the description of `ibconfig` in the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

Timing Errors

If your application fails, but the same calls issued interactively in the 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. This is only a problem with noncompliant GPIB devices.

A well-behaved IEEE 488 device does not experience timing errors. 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 NI-488.2 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 are controlled by the device and your application can adjust itself and work independently on any platform. Other delay mechanisms can exhibit differing behaviors on different platforms and thus might not eliminate timing errors.

Communication Errors

The following sections describe communication errors you might encounter in your application.

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-488.2 driver to perform repeat addressing if your device does not remain in its currently addressed state. You can either reconfigure from your application using `ibconfig`, or reconfigure using MAX.

Note National Instruments recommends using `ibconfig` to modify the configuration.

If your application uses `ibconfig`, it works properly regardless of the previous configuration. For more information about `ibconfig`, refer to the description of `ibconfig` in the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

Termination Method

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

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

Other Errors

If you experience other errors in your application, refer to the *NI-488.2 Help*. It includes extensive troubleshooting information and the answers to frequently asked questions. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

6

NI I/O Trace Utility

This chapter introduces you to NI I/O Trace, a utility that monitors and records multiple National Instruments APIs (for example, NI-488.2 and NI-VISA).

Overview

NI I/O Trace monitors, records, and displays the NI-488.2 calls made from NI-488.2 applications. You can use it to troubleshoot errors in your application and to verify the communication with your GPIB instrument. NI I/O Trace shows which NI-488.2 calls are being used to communicate with your instrument. If your application is not working properly, you can use NI I/O Trace to search for failed NI-488.2 calls.

Starting NI I/O Trace

Windows

To start NI I/O Trace, complete the following steps:

1. Start MAX as described in Chapter 2, *Measurement & Automation Explorer (Windows)*.
2. From the menu bar, select **Tools»NI I/O Trace**.

OS X and Linux

To start NI I/O Trace, complete the following steps:

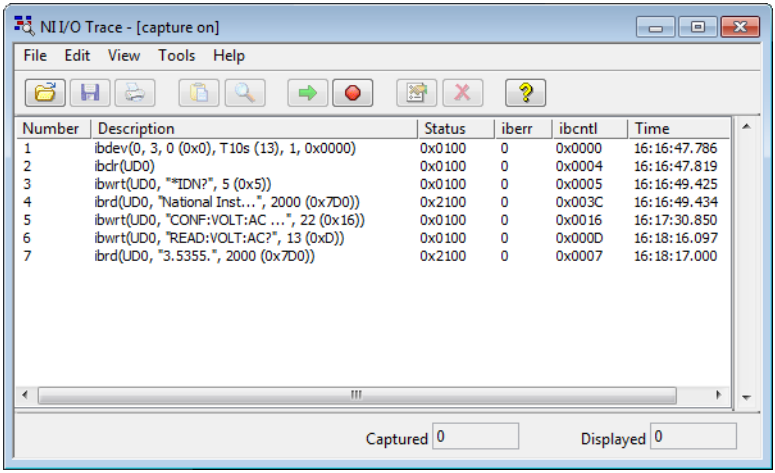
1. Start GPIB Explorer as described in Chapter 3, *GPIB Explorer (OS X and Linux)*.
2. From the menu bar, select **Tools»NI I/O Trace**.

Monitoring API Calls with NI I/O Trace

To display NI-488.2 API calls as they are made, do the following:

1. On the NI I/O Trace toolbar, click the green arrow button to start a capture.
2. Start the NI-488.2 application you want to monitor. NI I/O Trace records and displays all NI-488.2 calls, as shown in Figure 6-1.

Figure 6-1. NI-488.2 Calls Recorded by NI I/O Trace, Shown on Windows



Using the NI I/O Trace Help

To view the built-in, context-sensitive help for the NI I/O Trace utility, select **Help»Help Topics** in NI I/O Trace. You can also view the help by clicking on the question mark button on the NI I/O Trace toolbar, and then clicking on the area of the screen about which you have a question.

Locating Errors with NI I/O Trace

All NI-488.2 calls returned with an error are displayed in red within the main NI I/O Trace window.

Debugging Existing Applications

If the application does not have built-in error detection handling, you can use NI I/O Trace to determine which NI-488.2 call is failing.

Once you know which NI-488.2 call fails, refer to Appendix B, *Status Word Conditions*, and Appendix C, *Error Codes and Solutions*, for help understanding why the NI-488.2 call failed. This information is also available in the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

Viewing Properties for Recorded Calls

To see the detailed properties of any call recorded in the main NI I/O Trace window, double-click on the call. The **NI I/O Trace Property Sheet** window appears. It contains general, input, output, and buffer information, as applicable to each call.

Exiting NI I/O Trace

When you exit NI I/O Trace, its current configuration is saved and used to configure NI I/O Trace when you start it again. Unless you save the data captured in NI I/O Trace before you exit, that information is lost.

To save the captured data, stop the capture by clicking on the red button on the toolbar. Then, select **File»Save As** to save the data in a `.spy` file. After you save your data, select **File»Exit** to exit the NI I/O Trace utility.

Performance Considerations

NI I/O Trace can slow down the performance of your NI-488.2 application, and certain configurations of NI I/O Trace have a larger impact on performance than others. For example, configuring NI I/O Trace 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 your system. For this reason, use NI I/O Trace only while you are debugging your application or in situations where performance is not critical.

7

Interactive Control Utility

This chapter introduces you to the Interactive Control utility, which lets you communicate with GPIB devices interactively.

Overview

With the Interactive Control utility, you communicate with the GPIB devices through functions you interactively type in at the keyboard. For specific information about communicating with your particular device, refer to the documentation that came with the device. You can use the Interactive Control utility to practice communication with the instrument, troubleshoot problems, and develop your application.

The Interactive Control utility helps you to learn about your instrument and to troubleshoot problems by displaying the following information on your screen after you enter a command:

- Results of the status word (`Ibsta`) in hexadecimal notation
- Mnemonic constant of each bit set in `Ibsta`
- Mnemonic value of the error function (`Iberr`) if an error exists (the ERR bit is set in `Ibsta`)
- Count value for each read, write, or command function
- Data received from your instrument

Getting Started with Interactive Control

This section shows you how to use the Interactive Control utility to test a sequence of NI-488.2 calls.

For help on any Interactive Control command, type `help` followed by the command. For example, type `help ibdev` or `help devclear`.

To start the Interactive Control utility, complete the following steps:

(Windows)

1. Select **Start»Programs»National Instruments»Measurement & Automation** to start MAX.
2. Expand **Devices and Interfaces** and *Locate Your GPIB Interface*.

3. Right-click your GPIB interface and select **Interactive Control** from the drop-down menu that appears.

(OS X) Double-click **Applications»National Instruments»NI-488.2»Interactive Control**.

(Linux) Enter the following command:

/usr/local/bin/gpibintctrl

To use the Interactive Control utility to test a sequence of NI-488.2 calls, complete the following steps:

1. Open either an interface handle or device handle to use for further NI-488.2 calls. Use `ibdev` to open a device handle, `ibfind` to open an interface handle, or the `set 488.2` command to switch to a 488.2 prompt.

   The following example uses `ibdev` to open a device, assigns it to access interface `gpib0`, chooses a primary address of 6 with no secondary address, sets a timeout of 10 seconds (T10s = 13), enables the END message, and disables the EOS mode:

```
:ibdev
    enter board index: 0
    enter primary address: 6
    enter secondary address: 0
    enter timeout: 13
    enter 'EOI on last byte' flag: 1
    enter end-of-string mode/byte: 0
ud0:
```

Note If you type a command and no parameters, Interactive Control prompts you for the necessary arguments. If you already know the required arguments, you can type them at the command prompt, as follows:

```
:ibdev 0 6 0 13 1 0
ud0:
```

Note If you do not know the primary and secondary address of your GPIB instrument, use Interactive Control to discover it. First, select 488.2 style by entering `set 488.2 #` where `#` represents the board number (0 to 99) to which you have connected your device. Then use the `FindLstn` command to discover the address of your device. For help using `FindLstn`, enter `help findlstn` at the command prompt.

2. After you successfully complete `ibdev`, you have a `ud` prompt. The new prompt, `ud0`, represents a device-level handle that you can use for further NI-488.2 calls. To clear the device, use `ibclr`, as follows:

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

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

```
ud0: ibwrt
   enter string: "*IDN?"
[0100] (cmpl)
count: 5
```

Or, equivalently:

```
ud0: ibwrt "*IDN?"
[0100] (cmpl)
count: 5
```

4. To read data from your device, use `ibrd`. The data that is read from the instrument is displayed. For example, to read 29 bytes, enter the following:

```
ud0: ibrd
    enter byte count: 29
[0100] (cmpl)
count: 29
46 4C 55 4B 45 2C 20 34     FLUKE, 4
35 2C 20 34 37 39 30 31     5, 47901
37 33 2C 20 31 2E 36 20     73, 1.6
44 31 2E 30 0A              D.10.
```

Or, equivalently:

```
ud0: ibrd 29
[0100] (cmpl)
count: 29
46 4C 55 4B 45 2C 20 34     FLUKE, 4
35 2C 20 34 37 39 30 31     5, 47901
37 33 2C 20 31 2E 36 20     73, 1.6
44 31 2E 30 0A              D.10.
```

5. When you finish communicating with the device, make sure you put it offline using the `ibonl` command, as follows:

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

The `ibonl` command properly closes the device handle and the `ud0` prompt is no longer available.

6. To exit Interactive Control, type `q`.

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.
- Except for reads, the `count` parameter to calls is unnecessary because buffer lengths are automatically determined by Interactive Control.
- Function return values are handled automatically by Interactive Control. 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>. The Interactive Control utility then prompts you for each required parameter.

Number Syntax

You can enter numbers in either hexadecimal or decimal format.

Hexadecimal numbers—You must prefix hexadecimal numbers with `0x`. For example, `ibpad 0x16` sets the primary address to 16 hexadecimal (22 decimal).

Decimal numbers—Enter the number only. For example, `ibpad 22` sets the primary address to 22 decimal.

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 byte—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 `"*IDN?\r\n"`.

Address Syntax

Some of the NI-488.2 calls have an address or address list parameter. An address is a 16-bit representation of the GPIB device address. 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 `1,0xb706,3`.

Interactive Control Commands

Tables 7-1 and 7-2 summarize the syntax of the traditional NI-488.2 calls in the Interactive Control utility. Table 7-3 summarizes the syntax of the multi-device NI-488.2 calls in the Interactive Control utility. Table 7-4 summarizes the auxiliary functions that you can use in the Interactive Control utility. For more information about the function parameters type `help`. If you enter only the function name, the Interactive Control utility prompts you for parameters.

Table 7-1. Syntax for Device-Level Traditional NI-488.2 Calls in Interactive Control

Syntax	Description
`ibask option`	Return configuration information where `option` is a mnemonic for a configuration parameter.
`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.
`ibfind devname`	Return unit descriptor where `devname` is the symbolic name of the device template to use (for example, `dvm`).
`iblck v LockWaitTime`	Acquire or release an exclusive device lock for the current process.
`ibloc`	Go to local.
`ibnotify mask`	Start an asynchronous wait for selected events where mask is a hex or decimal integer or a list of mask bit mnemonics (for example, `ibnotify TIMO CMPL`).
`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.

Table 7-1. Syntax for Device-Level Traditional NI-488.2 Calls in Interactive Control (Continued)

Syntax	Description
`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 7-2. Syntax for Board-Level Traditional NI-488.2 Calls in 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.

Table 7-2. Syntax for Board-Level Traditional NI-488.2 Calls in Interactive Control (Continued)

Syntax	Description
`ibfind udname`	Return unit descriptor where `udname` is the symbolic name of interface (for example, `gpib0`).
`ibgts v`	Go from Active Controller to standby.
`ibist v`	Set/clear `ist`.
`iblck v LockWaitTime`	Acquire or release an exclusive interface lock for the current process.
`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.
`ibnotify mask`	Start an asynchronous wait for selected events where mask is a hex or decimal integer or a list of mask bit mnemonics (for example, `ibnotify TIMO CMPL`).
`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.

Table 7-2. Syntax for Board-Level Traditional NI-488.2 Calls in Interactive Control (Continued)

Syntax	Description
`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 7-3. Syntax for Multi-Device NI-488.2 Calls in Interactive Control

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.

Table 7-3. Syntax for Multi-Device NI-488.2 Calls in Interactive Control (Continued)

Syntax	Description
`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 7-4. Auxiliary Functions in Interactive Control

Function	Description
`set udname`	Select active device or interface where `udname` is the symbolic name of the new device or interface (for example, `dev1` or `gpib0`). Call `ibfind` or `ibdev` initially to open each device or interface.
`set 488.2 v`	Start using multi-device NI-488.2 calls for interface `v`.
`help`	Display the Interactive Control utility help.
`help option`	Display help information about `option`, where `option` is any NI-488.2 or auxiliary call (for example, `help ibwrt` or `help set`).
`!`	Repeat previous function.
–	Turn OFF display.
+	Turn ON display.

Table 7-4. Auxiliary Functions in Interactive Control (Continued)

Function	Description
`n * function`	Execute function `n` times where `function` represents the correct Interactive Control function syntax.
`n * !`	Execute previous function `n` times.
`$ filename`	Execute indirect file where `filename` is the pathname of a file that contains 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 the Interactive Control utility, all NI-488.2 calls (except `ibfind` and `ibdev`) 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 "*IDN?"
[0100] (cmpl)
count: 5
ud0:
```

For more information about `Ibsta`, refer to Appendix B, *Status Word Conditions*.

Error Information

If an NI-488.2 call completes with an error, the Interactive Control utility displays the relevant error mnemonic. In the following example, an error condition EBUS has occurred during a data transfer:

```
ud0: ibwrt "*IDN?"
[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 GPIB device is powered off or the GPIB cable is disconnected.

For a detailed list of the error codes and possible solutions, refer to Appendix C, *Error Codes and Solutions*.

Count Information

When an I/O function completes, the Interactive Control utility 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 to a multi-device NI-488.2 call is invalid, then the error is EARG and the Interactive Control utility displays the index of the invalid address as the count.

The count has a different meaning depending on which NI-488.2 call is made. For the correct interpretation of the count return, refer to the function descriptions in the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

8

NI-488.2 Programming Techniques

This chapter describes techniques for using some NI-488.2 calls in your application.

For more information about each function, refer to the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

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, EOI is asserted with the last byte of writes and the EOS modes are disabled.

You can use the `ibconfig` (`IbcEOT`) function to enable or disable the end of transmission (EOT) mode. If EOT mode is enabled, the GPIB EOI line is asserted 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 `ibconfig` (`IbcEOS`) 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 GPIB EOI line is automatically asserted 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, `ibrd`, `ibrda`, and `ibrdf` calls are terminated when the EOS byte is detected on the GPIB, when the GPIB EOI line is asserted, or when the specified count is reached. If the EOS read method is disabled, `ibrd`, `ibrda`, and `ibrdf` calls terminate only when the GPIB EOI line is asserted or the specified count has been read.

You can use the ibconfig function to configure the software to indicate whether 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, END is reported 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 part of the IEEE 488.1-2003 standard; thus, you can mix IEEE 488.1, IEEE 488.2, and HS488 devices in the same system. If HS488 is enabled, HS488-compliant interfaces implement high-speed transfers automatically when communicating with HS488 instruments. If you attempt to enable HS488 on a GPIB interface that does not have HS488-capable hardware, the ECAP error code is returned.

Enabling HS488

To enable HS488 for your GPIB interface, 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 interface for HS488, the Controller-In-Charge must send out GPIB command bytes (interface messages) to configure other devices for HS488 transfers.

If you are using device-level calls, the NI-488.2 software automatically sends the HS488 configuration message to devices. If you enabled the HS488 protocol in the GPIB Configuration utility, the NI-488.2 software sends out the HS488 configuration message when you use ibdev to bring a device online. When you use ibconfig on a board handle, the next device-level call will retransmit the cable length to the bus.

If you are using board-level traditional NI-488.2 calls or multi-device NI-488.2 calls 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 m, 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 more information about CFE and CFGn messages, refer to the *Multiline Interface Messages* topic in the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

System Configuration Effects on HS488

Maximum HS488 data transfer rates can be limited by your host computer and GPIB system setup. For example, when using a PC-compatible computer with PCI bus, the maximum obtainable transfer rate is 8 Mbytes/s, but when using another bus, such as USB or Ethernet, the maximum data transfer rate depends on the maximum transfer rate of that bus.

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 four HS488 devices connected by 4 m 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, 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 NI-488.2 Applications

NI-488.2 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 NI-488.2 driver automatically notifies your application that the event has occurred by invoking a callback function. The callback function is registered with the NI-488.2 driver when the `ibnotify` call is made.

Calling the ibnotify Function

`ibnotify` has the following function prototype:

```
unsigned int 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 NI-488.2 driver. If you are using device-level calls, you can 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 can call `ibnotify` with a board handle for `ud` and a `mask` of any values except RQS. 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 NI-488.2 driver when one or more of the mask bits passed to `ibnotify` is TRUE.

The callback function is of type `GPIBNotifyCallback_t` and is defined in the gpib header file, `ni4882.h`.

The callback function is passed a unit descriptor, the current values of the NI-488.2 global variables, and the user-defined reference data that was passed to the original `ibnotify` call. The NI-488.2 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 *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

Note The `ibnotify` callback is executed in a separate thread of execution from the rest of your application. If your application will be performing other NI-488.2 operations while it is using `ibnotify`, use the per-thread NI-488.2 globals that are provided by the `ThreadIbsta`, `ThreadIberr`, and `ThreadIbcnt` functions described in the *Writing Multithreaded NI-488.2 Applications* section. In addition, if your application needs to share global variables with the callback, use a synchronization primitive (for example, a semaphore) to protect access to any globals. For more information about the use of synchronization primitives, refer to the documentation about using operating system synchronization objects that came with your development tools.

ibnotify Programming Example

The following code is an example of how you can use `ibnotify` in your application. Assume that your GPIB device is a multimeter that you program to acquire a reading by sending `"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 only adds information (increases the count or adds another reading to the array of floats).

Note The following example is written using the `GpibNotifyCallback_t` definition for Windows. Refer to the gpib header file, `ni4882.h`, for the proper definition of the `Callback` thread for your platform. Other than a possible minor change in the definition of the `Callback` thread, this example will work on all platforms.

```
#include <stdio.h>
#include "ni4882.h"
int __stdcall MyCallback (int LocalUd, int LocalIbsta, int LocalIberr,
long LocalIbcntl, void *RefData);
int ReadingsTaken = 0;
float Readings[1000];
BOOL DeviceError = FALSE;
char expectedResponse = 0x43;

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 LocalIbsta, 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 (SpollByte != expectedResponse)   {
   printf ("Device returned invalid response. Status byte = 0x%x\n",
         SpollByte);
   DeviceError = TRUE;
   return 0;
}

// Read the data from the device. If the ERR bit is set in LocalIbsta,
// 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;
}

// The string returned by ibrd is a binary string whose length is
// specified by the byte count in ibcntl. However, many GPIB
// instruments return ASCII data strings and this example makes this
// assumption. Because of this, it is possible to add a NULL
// character to the end of the data received and use the printf()
// function to display the ASCII data. The following code
// illustrates that.
ReadBuffer[ibcntl] = '\0';

// Convert the data into a numeric value.
sscanf (ReadBuffer, "%f", &Readings[ReadingsTaken]);

// Display the data.
Printf ("Reading : %f\n", 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 NI-488.2 Applications

If you are writing a multithreaded NI-488.2 application and you plan to make all of your NI-488.2 calls from a single thread, you can safely continue to use the NI-488.2 global functions (`Ibsta`, `Iberr`, and `Ibcnt`). The NI-488.2 global functions are defined on a per-process basis, so each process accesses its own copy of the NI-488.2 globals.

If you are writing a multithreaded NI-488.2 application and you plan to make NI-488.2 calls from more than a single thread, you cannot safely continue to use the traditional NI-488.2 global functions without some form of synchronization (for example, semaphores, mutexes, critical sections). To understand why, refer to the following example.

Assume that a process has two separate threads that make NI-488.2 calls, thread 1 and thread 2. Just as thread 1 is about to examine one of the NI-488.2 globals, it gets preempted and thread 2 is allowed to run. Thread 2 proceeds to make several NI-488.2 calls that automatically update the NI-488.2 globals. Later, when thread 1 is allowed to run, the NI-488.2 global that it is ready to examine is no longer in a known state and its value is no longer reliable.

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

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

If you choose to implement the synchronization solution, you must ensure that the code making NI-488.2 calls and examining the NI-488.2 globals modified by a NI-488.2 call is protected by a synchronization primitive. For example, each thread might acquire a semaphore before making a NI-488.2 call and then release the semaphore after examining the NI-488.2 globals modified by the call. For more information about the use of synchronization primitives, refer to your operating system documentation about synchronization objects supported by your operating system.

If you choose not to use process-global functions, you can access per-thread copies of the NI-488.2 global variables using a special set of NI-488.2 calls. Whenever a thread makes an NI-488.2 call, the driver keeps a private copy of the NI-488.2 globals for that thread. The

following code shows the set of functions you can use to access these per-thread NI-488.2 global functions:

```
unsigned int ThreadIbsta(); // return thread-specific Ibsta()
unsigned int ThreadIberr(); // return thread-specific Iberr()
unsigned int ThreadIbcnt(); // return thread-specific Ibcnt()
```

In your application, instead of accessing the per-process NI-488.2 globals, substitute a call to get the corresponding per-thread NI-488.2 global. For example, you can replace the following line of code,

```
if (Ibsta() & ERR)
```

with

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

Note If you are using `ibnotify` in your application (refer to the *Asynchronous Event Notification in NI-488.2 Applications* section), the `ibnotify` callback is executed in a separate thread that is created by the NI-488.2 driver. Therefore, if your application makes NI-488.2 calls from the `ibnotify` callback function and makes NI-488.2 calls from other places, you must use the `ThreadIbsta`, `ThreadIberr`, and `ThreadIbcnt` functions described in this section, instead of the per-process NI-488.2 globals.

Device-Level Calls and Bus Management

The device-level traditional NI-488.2 calls are designed to perform all of the GPIB management for your application. However, the NI-488.2 driver can handle bus management only when the GPIB interface 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.

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

Talker/Listener Applications

Although designed for Controller-In-Charge applications, you can also use the NI-488.2 software in most non-Controller situations. These situations are known as Talker/Listener applications because the interface is not the GPIB Controller.

A Talker/Listener application typically uses `ibwait` with a mask of 0 to monitor the status of the interface. 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. 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 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 to 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 an 8-bit 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 the remaining seven 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

If you want your application to conduct a serial poll automatically when the SRQ line is asserted, you can enable automatic serial polling. The autopolling procedure occurs as follows:

1. To enable autopolling, use the board-level configuration function, `ibconfig`, with option `IbcAUTOPOLL`, or the GPIB Configuration utility. (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 requested service. 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 the response received. To prevent queue overflow, empty the queue as soon as an automatic serial poll occurs.
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, continue to call `ibrsp` until the RQS bit is cleared from the status word.

Stuck SRQ State

If autopolling is enabled and the GPIB interface detects an SRQ, the driver serial polls all open devices connected to that interface. 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 is enabled, the NI-488.2 software can perform autopolling after any device-level call provided that no GPIB I/O is currently in progress. Because the driver uses interrupts, an automatic serial poll can occur even when your application is not making any calls to the NI-488.2 software. Autopolling can also occur when a device-level `ibwait` for RQS is in progress. Autopolling is not allowed when an application calls a board-level traditional or multi-device NI-488.2 call, or the stuck SRQ (ESRQ) condition occurs.

SRQ and Serial Polling with Device-Level Traditional NI-488.2 Calls

You can use the device-level traditional NI-488.2 call `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 shows you how to use the `ibwait` and `ibrsp` functions in a typical SRQ servicing situation when automatic serial polling is enabled:

```
#include "ni4882.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 Multi-Device NI-488.2 Calls

The NI-488.2 software includes a set of multi-device NI-488.2 calls that you can use to conduct SRQ servicing and serial polling. Calls pertinent to SRQ servicing and serial polling are `AllSpoll`, `ReadStatusByte`, `FindRQS`, `TestSRQ`, and `WaitSRQ`. Following are descriptions of each of the calls:

- `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 (bit 6 or hex 40) 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 similar to the device-level NI-488.2 `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 call returns the index and status byte value of the device requesting service.
- `TestSRQ` determines whether the SRQ line is asserted 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 these calls 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.

Example 1: Using FindRQS

This example shows you how to use `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 shows you how to use `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 the traditional or multi-device NI-488.2 calls. If you use multi-device NI-488.2 calls to execute parallel polls, you do not need extensive knowledge of the parallel polling messages. However, you should use the traditional NI-488.2

calls for parallel polling when the GPIB interface is not the Controller, and the interface must configure itself for a parallel poll and set its own individual status bit (ist).

Parallel Polling with Traditional NI-488.2 Calls

Complete the following steps to implement parallel polling using traditional NI-488.2 calls. 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 the function configures 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 is 1, and 0 if the device is to assert the assigned data line when ist is 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 traditional NI-488.2 calls. The device asserts DIO7 if its ist is 0.

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

   ```
   #include "ni4882.h"
   dev = ibdev(0,3,0,T3s,1,0);
   ```

   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.

   ```
   ibppc(dev, 0x66);
   ```

   If the GPIB interface configures itself for a parallel poll, you should still use the ibppc function. Pass the interface index or an interface unit descriptor value as the first argument in ibppc. Also, if the individual status bit (ist) of the interface needs to be changed, use the IbcIst option with the ibconfig function.

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

   ```
   ibppc(0, 0x6C);
   ibconfig(0, IbcIst, 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 Multi-Device NI-488.2 Calls

Complete the following steps to implement parallel polling using multi-device NI-488.2 calls. Each step contains example code.

1. Configure the device for parallel polling using the `PPollConfig` call, 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 "ni4882.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` is 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 `PPollUnconfig`, 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);
```

A

GPIB Basics

The ANSI/IEEE Standard 488.1-2003, also known as the General Purpose Interface Bus (GPIB), describes a standard interface for communication between instruments and controllers from various vendors. It contains information about electrical, mechanical, and functional specifications. GPIB is a digital, 8-bit parallel communications interface that supports both interlocked and noninterlocked handshaking. The interlocked handshake, also known as three-wire handshake, allows for data transfer rates of 1 Mbyte/s and higher. The noninterlocked handshake, also known as HS488, allows for data transfer rates up to 8 Mbytes/s. 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 system has a National Instruments GPIB interface and 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 be either active or inactive (standby). Control can pass from the current CIC to an idle Controller, but only the System Controller, usually a GPIB interface, can make itself the CIC.

GPIB Addressing

All GPIB devices and interfaces 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 Controller uses this address to form a talk or listen address that is sent over the GPIB when communicating with a device.

Most devices just use primary addressing. The GPIB Controller manages the communication across the GPIB by using the addresses to designate which devices should be listening or talking at any given moment. Typically your computer is the GPIB Controller and it manages communication with your GPIB device by sending messages to it and receiving messages from it.

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. Table A-1 shows the configuration of the GPIB address bits.

Table A-1. GPIB Address Bits

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

With some devices, you can use secondary addressing. A secondary address is a number in the range hex 60 to hex 7E. When you use secondary addressing, 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 8 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 A-2 summarizes the GPIB handshake lines.

Table A-2. 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 (HS488) 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 hardware lines manage the flow of information across the bus. Table A-3 summarizes the GPIB interface management lines.

Table A-3. 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.

B

Status Word Conditions

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

For information about how to use `Ibsta()` or `ibsta` in your application program, refer to the *NI-488.2 Help*. For instructions on accessing the help, refer to the *Using the NI-488.2 Documentation* section of *About This Manual*.

Each bit in `Ibsta()` or `ibsta` can be set for device calls (dev), board calls (brd), or both (dev, brd). Table B-1 shows the status word layout.

Table B-1. Status Word Layout

Mnemonic	Bit Pos	Hex Value	Type	Description
ERR	15	8000	dev, brd	NI-488.2 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 function `Iberr`. Appendix C, *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 expired. TIMO is set in the status word following any synchronous I/O functions (for example, `ibcmd`, `ibrd`, `ibwrt`, `Receive`, `Send`, and `SendCmds`) if the timeout period expires before the I/O operation has completed. TIMO is also set in the status word following an `ibwait` or `ibnotify` call if the TIMO bit is set in the `mask` parameter and the timeout period expires before any other specified `mask` bit condition occurs. 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 interface 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 when the GPIB interface is CIC and the GPIB SRQ line is asserted. SRQI is cleared either when the GPIB interface 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 interface is in a lockout state. While LOK is set, the `EnableLocal` or `ibloc` call is inoperative for that interface. LOK is set whenever the GPIB interface detects that the Local Lockout (LLO) message has been sent either by the GPIB interface or by another Controller. LOK is cleared when the System Controller unasserts the Remote Enable (REN) GPIB line.

REM (brd)

REM indicates whether the interface is in the remote state. REM is set whenever the Remote Enable (REN) GPIB line is asserted and the GPIB interface detects that its listen address has been sent either by the GPIB interface or by another Controller. REM is cleared in the following situations:

- When REN becomes unasserted.
- When the GPIB interface as a Listener detects that the Go to Local (GTL) command has been sent either by the GPIB interface 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 interface is the Controller-In-Charge. CIC is set when the `SendIFC` or `ibsic` call is executed either while the GPIB interface is System Controller or when another Controller passes control to the GPIB interface. CIC is cleared either when the GPIB interface detects Interface Clear (IFC) from the System Controller or when the GPIB interface 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 interface is addressed as a Talker. TACS is set whenever the GPIB interface detects that its talk address (and secondary address, if enabled) has been sent either by the GPIB interface itself or by another Controller. TACS is cleared whenever the GPIB interface 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 interface is addressed as a Listener. LACS is set whenever the GPIB interface detects that its listen address (and secondary address, if enabled) has been sent either by the GPIB interface itself or by another Controller. LACS is also set whenever the GPIB interface shadow handshakes as a result of the `ibgts` function. LACS is cleared whenever the GPIB interface 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 interface has detected a device trigger command. DTAS is set whenever the GPIB interface, 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` or `ibnotify` call, if the DTAS bit is set in the `ibwait` mask parameter.

DCAS (brd)

DCAS indicates whether the GPIB interface has detected a device clear command. DCAS is set whenever the GPIB interface detects that the Device Clear (DCL) command has been sent by another Controller, or whenever the GPIB interface 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 NI-488.2 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 NI-488.2 call.

C

Error Codes and Solutions

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

Table C-1 lists the GPIB error codes.

Table C-1. GPIB Error Codes

Error Mnemonic	Value	Meaning
EDVR	0	System error
ECIC	1	Function requires GPIB interface to be CIC
ENOL	2	No Listeners on the GPIB
EADR	3	GPIB interface not addressed correctly
EARG	4	Invalid argument to function call
ESAC	5	GPIB interface not System Controller as required
EABO	6	I/O operation aborted (timeout)
ENEB	7	Nonexistent GPIB interface
EOIP	10	Asynchronous I/O in progress
ECAP	11	No capability for operation
EFSO	12	File system error
EBUS	14	GPIB bus error
ESRQ	16	SRQ stuck in ON position
ETAB	20	Table problem
ELCK	21	GPIB interface is locked and cannot be accessed
EARM	22	`ibnotify callback` failed to rearm
EHDL	23	Input handle is invalid
EWIP	26	Wait in progress on specified input handle
ERST	27	The event notification was cancelled due to a reset of the interface
EPWR	28	The interface lost power

EDVR (0)

EDVR is returned when the interface or device name passed to `ibfind`, or the interface index passed to `ibdev`, cannot be accessed. The global function `Ibcnt` contains an error code. This error occurs when you try to access an interface or device that is not installed or configured properly.

EDVR is also returned if there is an internal driver error.

Solutions

Possible solutions for this error are as follows:

- Use `ibdev` to open a device without specifying its symbolic name.
- Use only device or interface names that are configured in the GPIB Configuration utility as parameters to the `ibfind` function.
- Use troubleshooting tools to ensure that each interface you want to access is working properly:

  (Windows) Use the NI MAX Self-Test feature as described in Chapter 2, *Measurement & Automation Explorer (Windows)*.

  (OS X) Run **Applications»National Instruments»NI-488.2»Troubleshooting Wizard**.

  (Linux) Enter the following command:

  `/usr/local/bin/gpibtsw`
- Use the unit descriptor returned from `ibdev` or `ibfind` as the first parameter in subsequent traditional NI-488.2 calls. Examine the variable before the failing function to make sure its value has not been corrupted.

ECIC (1)

ECIC is returned when one of the following functions is called while the interface is not CIC:

- Any device-level traditional NI-488.2 calls that affect the GPIB.
- Any board-level traditional NI-488.2 calls that issue GPIB command bytes: `ibcmd`, `ibcmda`, `ibln`, and `ibrpp`.
- `ibcac` and `ibgts`.
- Any NI-488.2 multi-device calls that issue GPIB command bytes: `SendCmds`, `PPoll`, `Send`, and `Receive`.

Solutions

Possible solutions for this error are as follows:

- Use `ibsic` or `SendIFC` to make the GPIB interface become CIC on the GPIB.
- Use the `IbcSC` option in `ibconfig` to make sure your GPIB interface 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 interface.

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 interface is not the CIC and the Controller asserts ATN before the write call in progress has ended.

Solutions

Possible solutions for this error are as follows:

- 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 `ibconfig` with the `IbcPAD` (or `IbcSAD`, if necessary) options to match the configured address to the device switch settings.

EADR (3)

EADR occurs when the GPIB interface 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

Possible solutions for this error are as follows:

- Make sure that the GPIB interface 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:

- `ibconfig` with the `IbcTMO` option called with a value not in the range 0 through 17.
- `ibconfig` with the `IbcEOS` option called with meaningless bits set in the high byte of the second parameter.
- `ibconfig` with the `IbcPAD` or `IbcSAD` option called with invalid addresses.
- `ibppc` called with invalid parallel poll configurations.
- A multi-device NI-488.2 call made with an invalid address.
- `PPollConfig` called with an invalid data line or sense bit.

Solutions

Make sure that the parameters passed to the NI-488.2 call are valid.

ESAC (5)

ESAC results when `ibsic`, `ibsre`, `SendIFC`, `EnableRemote`, or the `IbcSRE` option in `ibconfig` is called when the GPIB interface does not have System Controller capability.

Solutions

Give the GPIB interface System Controller capability by calling the `IbcSC` option in `ibconfig` or by using MAX 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

Possible solutions for this error are as follows:

- 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 the `IbcTMO` option in `ibconfig`.
- Make sure that you have configured your device to send data before you request data.

ENEB (7)

ENEB occurs when a GPIB interface is configured for use by the system, but the driver cannot find the interface. This problem happens when the interface is not physically plugged into the system, the I/O address specified during configuration does not match the actual interface setting, or there is a system conflict with the base I/O address.

Solutions

Use troubleshooting tools to make sure there is a GPIB interface in your computer that is properly configured both in hardware and software and using a valid base I/O address:

(Windows) Use the NI MAX Self-Test feature as described in Chapter 2, *Measurement & Automation Explorer (Windows)*.

(OS X) Run **Applications»National Instruments»NI-488.2»Troubleshooting Wizard**.

(Linux) Enter the following command:

`/usr/local/bin/gpibtsw`

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 NI-488.2 calls. Resynchronization is accomplished by using one of the following functions:

`ibnotify callback`	If the `Ibsta` value passed to the `ibnotify` callback contains CMPL, the driver and application are resynchronized.
`ibnotify`	If the returned `Ibsta` contains CMPL, the driver and application are resynchronized.
`ibwait`	If the returned `Ibsta` contains CMPL, 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 interface 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 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.

Solutions

Possible solutions for this error are as follows:

- 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 `IbcTMO` option in the `ibconfig` function. EBUS results if a timeout occurred while sending these command bytes.

EBUS can occur if there are no functioning devices present on the GPIB.

Solutions

Possible solutions for this error are as follows:

- Verify that the instrument is operating correctly.
- Check your cable connections and make sure at least two-thirds of your devices are powered on.
- If the timeout period is too short for the driver to send command bytes, increase the timeout period.

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.

ELCK (21)

ELCK indicates that the requested operation could not be performed because of an existing lock by another process accessing the same interface. ELCK is also returned when a process attempts to unlock an interface for which it currently has no lock.

Solutions

Call `iblck` to lock the interface. If `iblck` continues to return ELCK, lengthen the `LockWaitTime` and wait for the other process to relinquish its interface lock.

Ensure that you have successfully locked the interface prior to unlocking it.

EARM (22)

EARM indicates that `ibnotify`'s asynchronous event notification mechanism failed to rearm itself. This generally occurs when an `ibnotify Callback` has attempted to rearm itself by returning an illegal value or when a fatal driver error (EDVR) has occurred.

Solutions

Ensure that the value being returned by your Callback function is a valid `ibnotify` mask value.

Return a zero value from your Callback function to unregister the asynchronous event notification mechanism. Then call `ibnotify` to re-enable notification.

EHDL (23)

EHDL results when an invalid handle is passed to a function call. The following are some examples:

- A valid board handle is passed in as a handle parameter to a device-level NI-488 function or a valid device handle is passed in as a handle parameter to a board-level NI-488 function.
- An invalid board or device unit descriptor is passed as input to any NI-488.2 function.
- A `boardID` outside the range of 0 to 99 is passed in to a traditional NI-488 board-level function or NI-488.2 routine.
- `ibconfig` or `ibask` is called with a device unit descriptor and a board-only configuration option, or with a board unit descriptor and a device-only configuration option.

Solutions

Do not use a device descriptor in a board function or vice-versa.

Make sure that the board index passed to the NI-488.2 call is valid.

EWIP (26)

EWIP indicates that an `ibwait` call is already in progress on the specified unit descriptor. This error occurs when one thread within a process calls `ibwait` on a given descriptor when another thread within the same process is already performing an `ibwait` using that same descriptor.

Solutions

Make sure that for any given unit descriptor only one thread calls `ibwait` at a time using that descriptor.

ERST (27)

ERST results when an event notification was cancelled due to a reset of the interface.

An `ibwait` call pending in the driver returns ERST in the following situations:

- Another thread in the same process calls `ibonl` using the same unit descriptor as ibwait.
- Another thread or another process issues a board-level `ibonl 1`.

An `ibnotify` Callback may be invoked with ERST in the following situations:

- Another process issues a board-level `ibonl 1`.

Solutions

Do not call `ibonl` with `ibwait` calls still pending in the driver.

Prevent other applications from calling `ibonl` by locking the interface with `iblck`.

EPWR (28)

EPWR results when an interface loses power. This often results when the system goes to and returns from a standby state.

Solutions

Take all handles offline and reinitialize the application.

Quit the application and restart.

Disable standby and hibernate modes on the PC.

D

Common Questions

This appendix answers some common questions about the NI-488.2 software.

General GPIB Questions

How many devices can I configure for use with the NI-488.2 software?

You can configure the NI-488.2 software to use up to 1,024 logical devices. The maximum number of physical devices you should connect to a single GPIB interface is 14, or fewer, depending on your system configuration.

When should I use the Interactive Control utility?

You can use the Interactive Control utility to test and verify instrument communication, troubleshoot problems, and develop your application. For more information, refer to Chapter 7, *Interactive Control Utility*.

How do I use an NI-488.2 application interface?

For information about using NI-488.2 application interfaces, refer to Chapter 4, *Developing Your NI-488.2 Application*.

What do I need to know to communicate properly with my GPIB instrument?

Refer to the documentation that came with your instrument. The command sequences that you use depend on the specific instrument. The documentation for each instrument should include the GPIB commands that you need to communicate with your instrument. In most cases, device-level traditional NI-488.2 calls are sufficient for communicating with instruments. For more information, refer to Chapter 4, *Developing Your NI-488.2 Application*.

How do I check for errors in my NI-488.2 application?

Examine the value of `Ibsta` after each 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 functions, refer to Chapter 5, *Debugging Your Application*.

How do I troubleshoot problems?
Use available tools to troubleshoot NI-488.2 problems:

- **(Windows)** Use the NI MAX Self-Test feature as described in Chapter 2, *Measurement & Automation Explorer (Windows)*.
- **(OS X)** Run **Applications»National Instruments»NI-488.2»Troubleshooting Wizard**.
- **(Linux)** Enter the following command:

  `/usr/local/bin/gpibtsw`

What information should I have before I call National Instruments?

Before you call National Instruments, record the results of running Self-Test (**Windows**) or the Troubleshooting Wizard (**OS X** and **Linux**).

How can I determine if my GPIB hardware and the NI-488.2 software are installed properly?

- **(Windows)** Use the NI MAX Self-Test feature as described in Chapter 2, *Measurement & Automation Explorer (Windows)*.
- **(OS X)** Run **Applications»National Instruments»NI-488.2»Troubleshooting Wizard**.
- **(Linux)** Enter the following command:

  `/usr/local/bin/gpibtsw`

How many GPIB interfaces can I configure for use with the NI-488.2 software?

You can configure the NI-488.2 software to communicate with up to 100 GPIB interfaces.

Windows

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

To view the NI-488.2 software version, complete the following steps:

1. Start MAX as described in Chapter 2, *Measurement & Automation Explorer (Windows)*.
2. Expand the **Software**.
3. Click **NI-488.2**.

   MAX displays the version number of the NI-488.2 software in the right window pane.

What do I do if my GPIB hardware is listed in the Windows Device Manager with a circled X or an exclamation point (!) overlaid on it?

Refer to the *Troubleshooting* topics in the *NI-488.2 Help* for information about what might cause this problem. If you cannot resolve the problem, contact National Instruments.

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

Start MAX as described in Chapter 2, *Measurement & Automation Explorer (Windows)*. Then expand **Devices and Interfaces**.

MAX lists your installed GPIB hardware under **Devices and Interfaces** and *Locate Your GPIB Interface*.

Are interrupts and DMA required for the NI-488.2 software?

Generally, plug-in interface cards, such as the PCI-GPIB, require interrupt resources in your computer. Remote interfaces, such as the GPIB-USB and GPIB Ethernet products, do not require interrupt resources in your computer. There may be exceptions to this statement. Refer to the general readme file located on your installation media or in the installation directory, for the latest interface options supported by the current version of NI-488.2.

DMA is not required for the NI-488.2 software.

E

NI Services

National Instruments provides global services and support as part of our commitment to your success. Take advantage of product services in addition to training and certification programs that meet your needs during each phase of the application life cycle; from planning and development through deployment and ongoing maintenance.

To get started, register your product at ni.com/myproducts.

As a registered NI product user, you are entitled to the following benefits:

- Access to applicable product services.
- Easier product management with an online account.
- Receive critical part notifications, software updates, and service expirations.

Log in to your National Instruments ni.com User Profile to get personalized access to your services.

Services and Resources

- **Maintenance and Hardware Services—**NI helps you identify your systems' accuracy and reliability requirements and provides warranty, sparing, and calibration services to help you maintain accuracy and minimize downtime over the life of your system. Visit ni.com/services for more information**.**
  - **Warranty and Repair—**All NI hardware features a one-year standard warranty that is extendable up to five years. NI offers repair services performed in a timely manner by highly trained factory technicians using only original parts at a National Instruments service center.
  - **Calibration—**Through regular calibration, you can quantify and improve the measurement performance of an instrument. NI provides state-of-the-art calibration services. If your product supports calibration, you can obtain the calibration certificate for your product at ni.com/calibration.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.

- **Training and Certification**—The NI training and certification program is the most effective way to increase application development proficiency and productivity. Visit ni.com/training for more information.
  - The Skills Guide assists you in identifying the proficiency requirements of your current application and gives you options for obtaining those skills consistent with your time and budget constraints and personal learning preferences. Visit ni.com/skills-guide to see these custom paths.
  - NI offers courses in several languages and formats including instructor-led classes at facilities worldwide, courses on-site at your facility, and online courses to serve your individual needs.
- **Technical Support**—Support at ni.com/support includes the following resources:
  - **Self-Help Technical Resources**—Visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Software Support Service Membership**—The Standard Service Program (SSP) is a renewable one-year subscription included with almost every NI software product, including NI Developer Suite. This program entitles members to direct access to NI Applications Engineers through phone and email for one-to-one technical support, as well as exclusive access to online training modules at ni.com/self-paced-training. NI also offers flexible extended contract options that guarantee your SSP benefits are available without interruption for as long as you need them. Visit ni.com/ssp for more information.
- **Declaration of Conformity (DoC)**—A DoC is our claim of compliance with the Council of the European Communities using the manufacturer's declaration of conformity. This system affords the user protection for electromagnetic compatibility (EMC) and product safety. You can obtain the DoC for your product by visiting ni.com/certification.

For information about other technical support options in your area, visit ni.com/services, or contact your local office at ni.com/contact.

You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office websites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Glossary

Symbol	Prefix	Value
p	pico	10^{-12}
n	nano	10^{-9}
µ	micro	10^{-6}
m	milli	10^{-3}
k	kilo	10^{3}
M	mega	10^{6}
G	giga	10^{9}
T	tera	10^{12}

A

acceptor handshake — Listeners use this GPIB interface function to receive data, and all devices use it to receive commands. *See also* 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.

API — Application Programming Interface.

application interface — Formerly called *language interface*. Code that enables an application program that uses NI-488.2 calls to access the driver.

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 — A feature of the GPIB software in which serial polls are executed automatically by the driver whenever a device asserts the GPIB SRQ line. Also called autopolling.

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 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 also* handshake.
DCL	Device Clear—The GPIB command used to reset the device or internal functions of all devices. *See also* 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 also* 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 signals 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—Part of the IEEE 488.2-defined status byte which is received from a device responding to a serial poll.

F

FIFO	First-in-first-out.

G

GET	Group Execute Trigger—The GPIB command used to trigger a device or internal function of an addressed Listener.
GPIB	General Purpose Interface Bus—This is the common name for the communications interface system defined in ANSI/IEEE Standard 488.1-2003 and ANSI/IEEE Standard 488.2-1992.
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 interfaces.

GTL — Go To Local—The GPIB command used to place an addressed Listener in local (front panel) control mode.

H

handshake — The mechanism used to transfer bytes from the source handshake function of one device to the acceptor handshake function of another device. DAV, NRFD, and NDAC, three GPIB lines, 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 488.1-2003.

hex — Hexadecimal—A number represented in base 16. For example, decimal 16 is hex 10.

high-level function — *See* device-level function.

HS488 — A high-speed data transfer protocol for IEEE 488. This protocol increases performance for GPIB reads and writes up to 8 Mbytes/s, depending on your system.

Hz — Hertz.

I

I/O — Input/output—In this manual, it is the transmission of commands or messages between the system 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.

`ibcnt` — After each NI-488.2 I/O call, this global variable contains the actual number of bytes transmitted. On systems with a 16-bit integer, such as MS-DOS, `ibcnt` is a 16-bit integer, and `ibcntl` is a 32-bit integer. For cross-platform compatibility, use `ibcntl`, unless using the newer NI4882 API. For accessing the newer NI4882 API, use the global function, `Ibcnt`, instead.

Ibcnt	After each NI-488.2 call, this global function contains the actual number of bytes transmitted. The Ibcnt function returns a 32-bit integer. For accessing the newer NI4882 API, this function is recommended instead of the global variables, ibcnt and ibcntl.
ibcntl	After each NI-488.2 I/O call, this global variable contains the actual number of bytes transmitted. On systems with a 16-bit integer, such as MS-DOS, ibcnt is a 16-bit integer, and ibcntl is a 32-bit integer. For cross-platform compatibility, use ibcntl, unless using the newer NI4882 API. For accessing the newer NI4882 API, use the global function, Ibcnt, instead.
iberr	A global variable that contains the specific error code associated with a function call that failed. For accessing the newer NI4882 API, use the global function, Iberr, instead.
Iberr	A global function that contains the specific error code associated with a function call that failed. For accessing the newer NI4882 API, this function is recommended instead of the global variable, iberr.
ibsta	At the end of each function call, this global variable (status word) contains status information. For accessing the newer NI4882 API, use the global function, Ibsta, instead.
Ibsta	At the end of each function call, this global function contains status information. For accessing the newer NI4882 API, this function is recommended instead of the global variable, ibsta.
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.
ISA	Industry Standard Architecture.
ist	An Individual Status bit of the status byte used in the Parallel Poll Configure function.

L

LAD	Listen Address. *See also* MLA.
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.
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 also* handshake.
NRFD	Not Ready For Data—One of the three GPIB handshake lines. *See also* handshake.

P

parallel poll	The process of polling all configured devices at once and reading a composite poll response. *See also* serial poll.
PC	Personal computer.
PCI	Peripheral Component Interconnect.
PIO	*See* programmed I/O.
PPC	Parallel Poll Configure—The GPIB command used to configure an addressed Listener to participate in polls.
PPD	Parallel Poll Disable—The GPIB command used to disable a configured device from participating in polls. There are 16 PPD commands.
PPE	Parallel Poll Enable—The GPIB command used to enable a configured device to participate in polls and to assign a DIO response line. There are 16 PPE commands.
PPU	Parallel Poll Unconfigure—The GPIB command used to disable any device from participating in polls.
programmed I/O	Low-speed data transfer between the GPIB interface and memory in which the CPU moves each data byte according to program instructions. *See also* DMA.

R

resynchronize	When the driver indicates to the application that 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 also* 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 also* 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 also* 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 also* 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 GPIB 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 also* 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 GPIB 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 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

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

V

W