NI-488.2™
User Manual for Macintosh

January 1995 Edition

Part Number 320897A-01

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

Branch Offices:
Australia (03) 879 9422, Austria (0662) 435986, Belgium 02/757.00.20,
Canada (Ontario) (519) 622-9310, Canada (Québec) (514) 694-8521,
Denmark 45 76 26 00, Finland (90) 527 2321, France (1) 48 14 24 24,
Germany 089/741 31 30, Italy 02/48301892, Japan (03) 3788-1921,
Mexico 95 800 010 0793, Netherlands 03480-33466, Norway 32-84 84 00,
Singapore 2265886, Spain (91) 640 0085, Sweden 08-730 49 70,
Switzerland 056/20 51 51, Taiwan 02 377 1200, U.K. 0635 523545

Limited Warranty

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

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

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

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

Copyright

Trademarks

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

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

Contents

About This Manual xi
How to Use This Manual Set xi
Organization of This Manual xii
Conventions Used in This Manual xiii
Related Documentation xiii
Customer Communication xiv

Chapter 1
Introduction 1-1
GPIB Overview 1-1
Talkers, Listeners, and Controllers 1-1
Controller-In-Charge and System Controller 1-1
GPIB Addressing 1-2
Sending Messages Across the GPIB 1-2
Data Lines 1-2
Handshake Lines 1-3
Interface Management Lines 1-3
Setting Up and Configuring Your System 1-4
Controlling More Than One Board 1-5
Configuration Requirements 1-5
The NI-488.2 Software Components 1-6
NI-488.2 Driver and Driver Utilities 1-6
C Language Files 1-7
QuickBASIC Language Files 1-7
Device Manager Files 1-8
How the NI-488.2 Software Works with Your System 1-8

Chapter 2
Developing Your Application 2-1
Choosing a Programming Method 2-1
Using the NI-488.2 Language Interface 2-1
Using NI-488 Functions: One Device for Each Board 2-1
NI-488 Device Functions 2-2
NI-488 Board Functions 2-2
Using NI-488.2 Routines: Multiple Boards and/or Multiple Devices 2-2
Using the Device Manager 2-3
Checking Status with Global Variables 2-3
Status Word – ibsta 2-3
Error Variable – iberr 2-5
Count Variables – ibcnt and ibcntl 2-5
Using IBIC 488.2 to Communicate with Devices 2-5
Writing Your NI-488 Application 2-6
Items to Include 2-6

NI-488 Program Shell 2-7
General Program Steps and Examples 2-8
Step 1. Open a Device 2-8
Step 2. Clear the Device 2-8
Step 3. Configure the Device 2-8
Step 4. Trigger the Device 2-9
Step 5. Wait for the Measurement 2-9
Step 6. Read the Measurement 2-10
Step 7. Process the Data 2-10
Step 8. Place the Device Offline 2-10
Writing Your NI-488.2 Application 2-11
Items to Include 2-11
NI-488.2 Program Shell 2-12
General Program Steps and Examples 2-13
Step 1. Initialization 2-13
Step 2. Find All Listeners 2-13
Step 3. Identify the Instrument 2-13
Step 4. Initialize the Instrument 2-14
Step 5. Configure the Instrument 2-15
Step 6. Trigger the Instrument 2-15
Step 7. Wait for the Measurement 2-15
Step 8. Read the Measurement 2-16
Step 9. Process the Data 2-16
Step 10. Place the Board Offline 2-16
Compiling, Linking, and Running 2-17
C Applications 2-17
QuickBASIC Applications 2-17

Chapter 3
Debugging Your Application 3-1
Running NI-488.2 Test 3-1
Debugging with the Global Status Variables 3-1
Debugging with IBIC 488.2 3-1
GPIB Error Codes 3-1
Configuration Errors 3-2
Timing Errors 3-3
Communication Errors 3-3
Repeat Addressing 3-3
Termination Method 3-4
Common Questions 3-4

Chapter 4
Interface Bus Interactive Control Utility 4-1
Overview 4-1
Example Using NI-488 Functions 4-1
IBIC 488.2 Syntax 4-4
Number Syntax 4-4
String Syntax 4-4
Address Syntax 4-5
IBIC 488.2 Syntax for NI-488 Functions 4-5
IBIC 488.2 Syntax for NI-488.2 Routines 4-8
Status Word 4-9
Error Information 4-9
Count 4-9
Common NI-488 Functions 4-10
ibfind 4-10
ibdev 4-10
ibwrt 4-12
ibrd 4-12
Common NI-488.2 Routines in IBIC 488.2 4-12
Set 4-12
Send and SendList 4-13
Receive 4-13
Auxiliary Functions 4-14
Set (Select Device or Board) 4-14
Help (Display Help Information) 4-15
! (Repeat Previous Function) 4-15
- (Turn Display Off) and + (Turn Display On) 4-15
n* (Repeat Function n Times) 4-16
$ (Execute Indirect File) 4-16
Print (Display the ASCII String) 4-16
Buffer (Set Buffer Display Mode) 4-17

Chapter 5
GPIB Programming Techniques 5-1
Termination of Data Transfers 5-1
High-Speed Data Transfers (HS488) 5-2
Enabling HS488 5-2
System Configuration Effects on HS488 5-3
Waiting for GPIB Conditions 5-3
Device-Level Calls and Bus Management 5-3
Talker/Listener Applications 5-4
Waiting for Messages from the Controller 5-4
Requesting Service 5-4
Serial Polling 5-5
Service Requests from IEEE 488 Devices 5-5
Service Requests from IEEE 488.2 Devices 5-5
Automatic Serial Polling 5-5

Stuck SRQ State 5-6
Autopolling and Interrupts 5-6
C “ON SRQ” Capability 5-6
SRQ and Serial Polling with NI-488 Device Functions 5-7
SRQ and Serial Polling with NI-488.2 Routines 5-7
Example 1: Using FindRQS 5-8
Example 2: Using AllSpoll 5-9
Parallel Polling 5-9
Implementing a Parallel Poll 5-9
Parallel Polling with NI-488 Functions 5-10
Parallel Polling with NI-488.2 Routines 5-11

Chapter 6
GPIB Configuration Utility 6-1
Overview 6-1
Running the Configuration Utility 6-1
Opening the Configuration Utility 6-1
Default Configuration 6-3
Control Items 6-4
Help Frame 6-5
Global Frame 6-6
Bus/Device Frame 6-7
Options for Buses or Devices 6-8
Primary Address 6-8
Secondary Address 6-8
Timeout 6-8
EOS Modes 6-9
EOS Byte 6-9
Options for Buses Only 6-9
Bus Timing 6-9
TNT High Speed 6-10
DMA 6-10
System Controller 6-10
Assert REN when System (Controller) 6-10
Unaddressing 6-10
Repeat Addressing 6-11
Options for Devices Only 6-11
Rename Device 6-11
Use Bus 6-11
Exiting the Configuration Utility 6-11

Appendix A
Status Word Conditions A-1

Appendix B
Error Codes and Solutions B-1

Appendix C
Device Manager Interface C-1

Appendix D
Customer Communication D-1

Glossary G-1

Index I-1

Figures

Figure 1-1. GPIB Address Bits 1-2
Figure 1-2. Linear and Star System Configuration 1-4
Figure 1-3. Example of Multiboard System Setup 1-5
Figure 1-4. How the NI-488.2 Software Works with Your System 1-8

Figure 2-1. General Program Shell Using NI-488 Device Functions 2-7
Figure 2-2. General Program Shell Using NI-488.2 Routines 2-12

Figure 6-1. Opening Screen of NI-488 Config 6-2
Figure 6-2. Device Default Settings in NI-488 Config 6-3
Figure 6-3. Help Frame in NI-488 Config 6-5
Figure 6-4. Manual Bus Association in NI-488 Config 6-6

Tables

Table 1-1. GPIB Handshake Lines 1-3
Table 1-2. GPIB Interface Management Lines 1-3

Table 2-1. Status Word (ibsta) Layout 2-4

Table 3-1. GPIB Error Codes 3-2

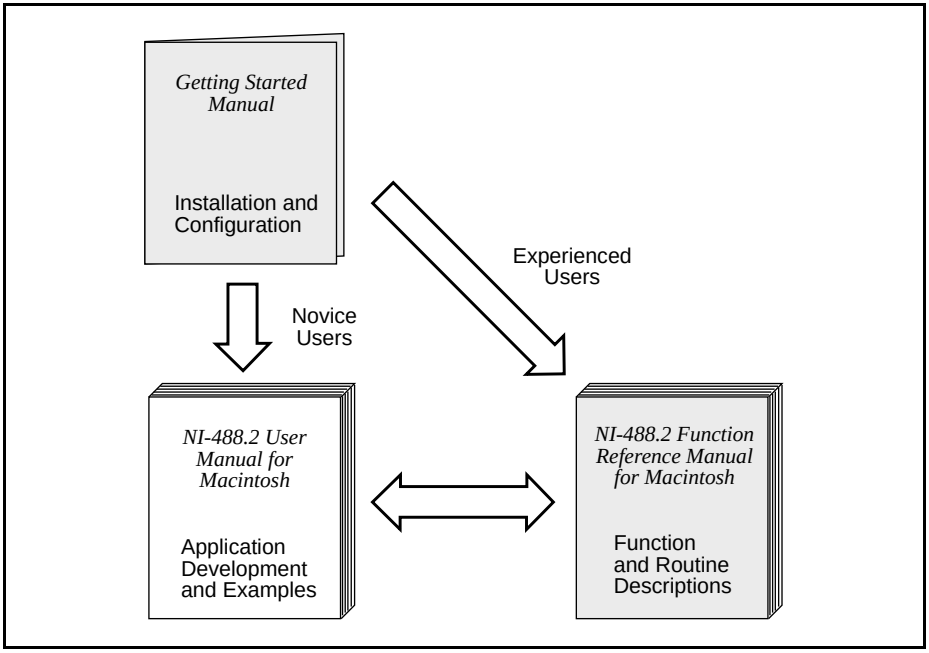
Table 4-1. Syntax for Device-Level NI-488 Functions in IBIC 488.2 4-6
Table 4-2. Syntax for Board-Level NI-488 Functions in IBIC 488.2 4-7
Table 4-3. Syntax for NI-488.2 Routines in IBIC 488.2 4-8
Table 4-4. Auxiliary Functions in IBIC 488.2 4-14

Table 6-1. Bus/Device Options in NI-488 Config 6-7

About This Manual

This manual describes the features and functions of the NI-488.2 software for Macintosh. This manual assumes that you are already familiar with the Macintosh operating system.

How to Use This Manual Set



Use the getting started manual that came with your kit to install and configure your GPIB hardware and NI-488.2 software.

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

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

Organization of This Manual

This manual is organized as follows:

- Chapter 1, *Introduction*, gives an overview of GPIB and the NI-488.2 software.
- Chapter 2, *Developing Your Application*, explains how to develop a GPIB application program using NI-488 functions and NI-488.2 routines.
- Chapter 3, *Debugging Your Application,* describes several ways to debug your application program.
- Chapter 4, *Interface Bus Interactive Control Utility*, introduces you to `IBIC 488.2`, the interactive control utility you can use to communicate with GPIB devices interactively.
- Chapter 5, *GPIB Programming Techniques*, describes techniques for using some NI-488 functions and NI-488.2 routines in your application program.
- Chapter 6, *GPIB Configuration Utility*, contains instructions for configuring the NI-488.2 software with the `NI-488 Config` utility.
- Appendix A, *Status Word Conditions*, gives a detailed description of the conditions reported in the status word, `ibsta`.
- Appendix B, *Error Codes and Solutions*, lists a description of each error, some conditions under which it might occur, and possible solutions.
- Appendix C, *Device Manager Interface*, contains information for programming your GPIB interface from any language using the Device Manager functions.
- Appendix D, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.
- The *Index* contains an alphabetical list of key terms and topics in this manual, including the page where you can find each one.

Conventions Used in This Manual

The following conventions are used in this manual.

bold	Bold text denotes commands, menus, menu items, options, and screen button names and checkboxes.
italic	Italic text denotes emphasis, cross references, field names, or an introduction to a key concept.
bold italic	Bold italic text denotes a note, caution, or warning.
`monospace`	Text in this font denotes text or characters that you enter from the keyboard. Sections of code, programming examples, and syntax examples also appear in this font. This font is also used for the proper name of disk drives, paths, directories, device names, variables, and for statements taken from program code.
`bold monospace`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen.
`italic monospace`	Italic text in this font denotes that you must supply the appropriate words or values in the place of these items.
<>	Angle brackets enclose the name of a key on the keyboard—for example, <Shift>.
IEEE 488 and IEEE 488.2	*IEEE 488* and *IEEE 488.2* refer to the ANSI/IEEE Standard 488.1-1987 and the ANSI/IEEE Standard 488.2-1987, respectively, which define the GPIB.

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

Related Documentation

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

- ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation*
- ANSI/IEEE Standard 488.2-1987, *IEEE Standard Codes, Formats, Protocols, and Common Commands*
- *Inside Macintosh*, Apple Computer, Inc., Reading, MA, 1987

- *Macintosh Programmer's Workshop, Version 3.3*, Apple Computer, Inc., Cupertino, CA, 1993
- *Metrowerks CodeWarrior User's Guide*, Metrowerks, Inc., Mooers, NY
- *Microsoft QuickBASIC*, Microsoft Corp., Redmond, WA, 1988
- *THINK C User's Manual*, Symantec Corp., Bedford, MA

Customer Communication

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

Chapter 1
Introduction

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

GPIB Overview

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

Talkers, Listeners, and Controllers

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

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

Controller-In-Charge and System Controller

You can have multiple Controllers on the GPIB, but only one Controller at a time can be the active Controller, or Controller-In-Charge (CIC). When a Controller is not active, it is considered an idle Controller. Active control can pass from the current CIC to an idle Controller. The System Controller, usually a GPIB interface board, is the only device on the bus that can make itself the CIC.

GPIB Addressing

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

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

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

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

Figure 1-1. GPIB Address Bits

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

Sending Messages Across the GPIB

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

Data Lines

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

Handshake Lines

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

Table 1-1. GPIB Handshake Lines

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

Interface Management Lines

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

Table 1-2. GPIB Interface Management Lines

Line	Description
ATN (attention)	Controller drives ATN true when it sends commands and false when it sends data messages.
IFC (interface clear)	System Controller drives the IFC line to initialize the bus and make itself CIC.
REN (remote enable)	System Controller drives the REN line to place devices in remote or local program mode.
SRQ (service request)	Any device can drive the SRQ line to asynchronously request service from the Controller.
EOI (end or identify)	Talker uses the EOI line to mark the end of a data message. Controller uses the EOI line when it conducts a parallel poll.

Setting Up and Configuring Your System

Devices are usually connected with a cable assembly consisting of a shielded 24-conductor cable with both a plug and receptacle connector at each end. With this design, you can link devices in a linear configuration, a star configuration, or a combination of the two. Figure 1-2 shows the linear and star configurations.

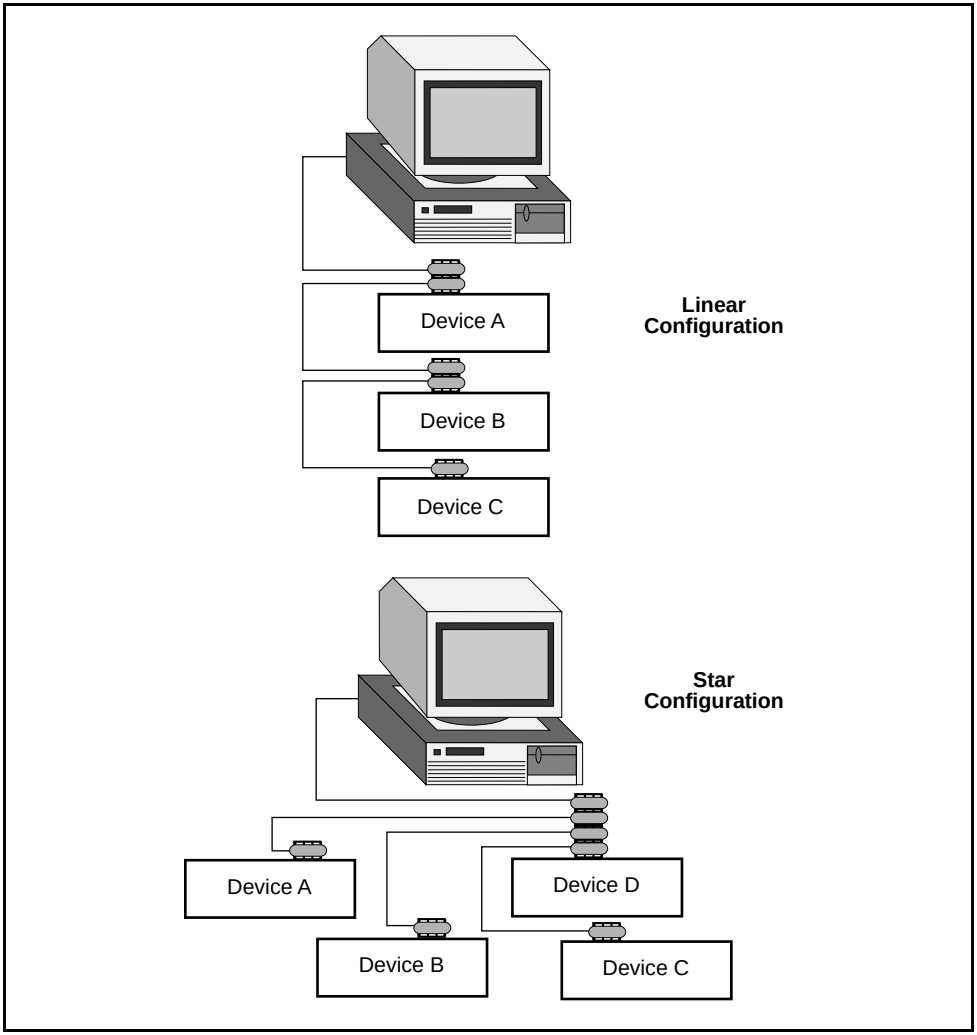


Figure 1-2. Linear and Star System Configuration

Controlling More Than One Board

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

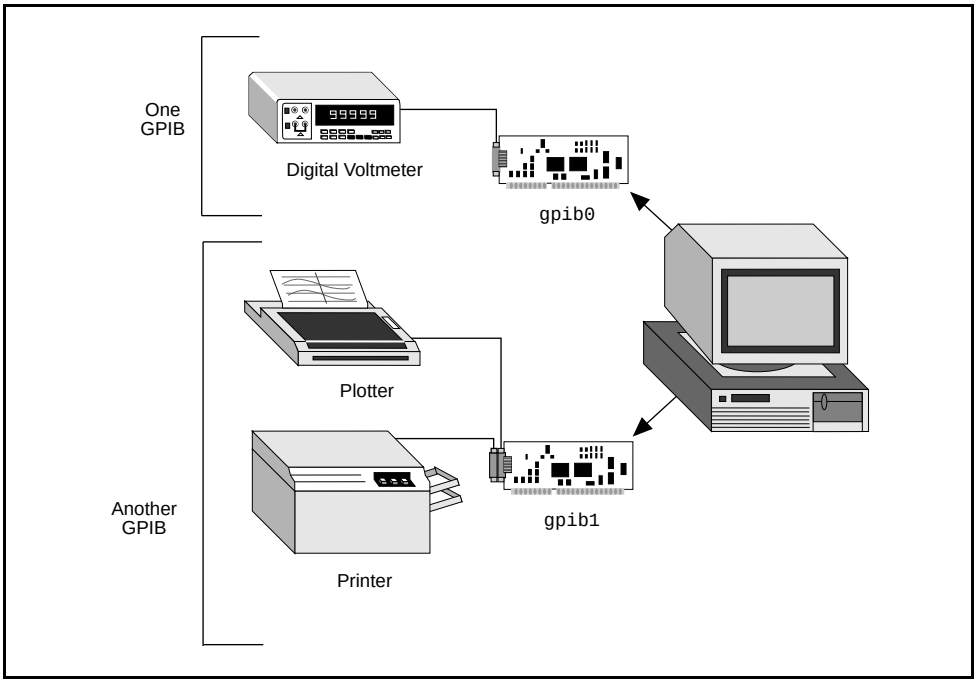


Figure 1-3. Example of Multiboard System Setup

Configuration Requirements

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

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

For high-speed operation, the following restrictions apply:

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

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

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

NI-488.2 Software Components

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

NI-488.2 Driver and Driver Utilities

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

- `Read Me` is a documentation file that contains important information about the NI-488.2 software and a description of any new features. Before you use the software, read this file for the most recent information.
- `NI-488.2 Installer` is an application that installs the NI-488.2 software.
- `NI-488 INIT` loads the appropriate drivers for installed National Instruments GPIB interfaces. The `NI-488 INIT` is loaded into memory when the Macintosh is booted.
- `NI-488 Config` is a configuration utility that you can use to examine or change the software settings.
- `NB-Boards` is a configuration utility that displays information about the boards currently installed in your computer if it contains plug-in slots.
- `NI-488.2 Test` is a software diagnostic utility.
- `IBIC 488.2` is an interactive control program that you use to communicate with the GPIB devices interactively using NI-488.2 functions and routines. It helps you to learn the NI-488.2 routines and to program your instrument or other GPIB devices.

- `NI-DMA/DSP` is a system extension that provides DMA functionality through an RTSI connection to an NB-DMA2800 or NB-DMA-8.
- The `Ethernet` folder contains utilities that are applicable if you have a National Instruments GPIB-ENET.

C Language Files

The C LI Folder contains files relevant to programming in THINK C, MPW C, and Metrowerks CodeWarrior C.

- `decl.h` is a file containing useful variable and constant declarations.
- `LI.c` is a file containing THINK C , MPW C, and Metrowerks CodeWarrior C language interface code.
- `DrInterface.h` is a file containing declarations of structures used and should be included in your program.
- `dcsamp.make` is the C make file for MPW.
- `dcsamp.c` is a sample program using device calls.
- `bcsamp.c` is a sample program using board calls.

QuickBASIC Language Files

The BASIC LI Folder contains a library, initialization code, and examples.

- `QuickBASIC4882.lib` is a library file loaded by your QuickBASIC program.
- `QB488Init.bas` is a file which must be merged at the beginning of your program.
- `QB488Voc.bas` is an example of each call in the QuickBASIC NI-488 vocabulary.
- `QB4882Voc.bas` is an example of each call in the QuickBASIC NI-488.2 vocabulary.
- `Dbsamp.bas` is a sample program using device calls.
- `Bbsamp.bas` is a sample program using board calls.
- `QBSAMP4882.bas` is a sample program using NI-488.2 calls.

Device Manager Files

The Device Manager calls folder includes the following two files.

- `controlcalls.c` is a sample program making high-level Device Manager calls.
- `pbcontrolcalls.c` is a sample program making low-level Device Manager calls.

How the NI-488.2 Software Works with Your System

The NI-488.2 driver is a device driver that is loaded at system startup.

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

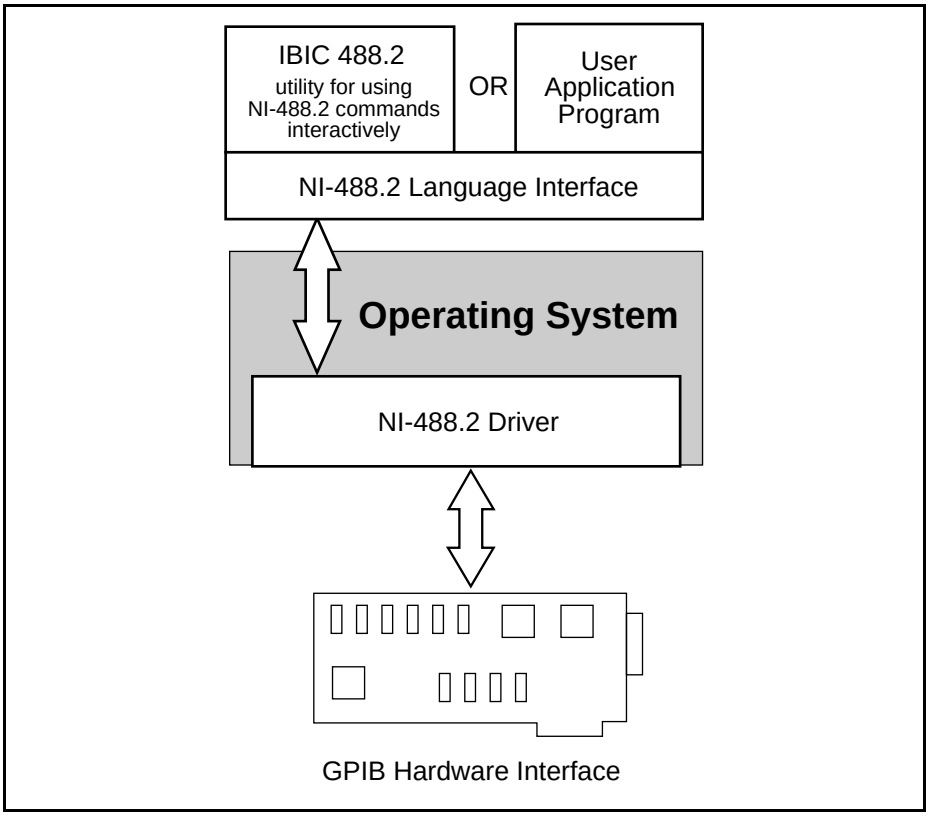


Figure 1-4. How the NI-488.2 Software Works with Your System

Chapter 2
Developing Your Application

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

Choosing a Programming Method

Programs that need to communicate across the GPIB can access the NI-488.2 driver using either the NI-488.2 language interface or the Device Manager interface.

Using the NI-488.2 Language Interface

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

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

Using NI-488 Functions: One Device for Each Board

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

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

The NI-488 functions consist of high-level (or device) functions that hide much of the GPIB management operations and low-level (or board) functions that offer you more control over the GPIB than NI-488.2 routines. The following sections describe these different function types.

NI-488 Device Functions

Device functions are high-level functions that automatically execute commands that handle bus management operations such as reading from and writing to devices or polling them for status. If you use device functions, you do not need to understand GPIB protocol or bus management. For information about device-level calls and how they manage the GPIB, refer to *Device-Level Calls and Bus Management* in Chapter 5, *GPIB Programming Techniques.*

NI-488 Board Functions

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

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

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

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

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

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

Using the Device Manager

You might want to use the Device Manager interface if your application requires true asynchronous calls or completion routines. Refer to Appendix C, *Device Manager Interface*, for more information. Using the NI-488.2 language interfaces is the recommended programming method.

Checking Status with Global Variables

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

Status Word – ibsta

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

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

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

Table 2-1. Status Word (ibsta) Layout

Mnemonic	Bit Pos.	Hex Value	Type	Description
ERR	15	8000	dev, brd	GPIB error
TIMO	14	4000	dev, brd	Time limit exceeded
END	13	2000	dev, brd	END or EOS detected
SRQI	12	1000	brd	SRQ interrupt received
RQS	11	800	dev	Device requesting service
CMPL	8	100	dev, brd	I/O completed
LOK	7	80	brd	Lockout State
REM	6	40	brd	Remote State
CIC	5	20	brd	Controller-In-Charge
ATN	4	10	brd	Attention is asserted
TACS	3	8	brd	Talker
LACS	2	4	brd	Listener
DTAS	1	2	brd	Device Trigger State
DCAS	0	1	brd	Device Clear State

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

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

or

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

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

Error Variable – iberr

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

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

For more information on error codes and solutions refer to Chapter 3, *Debugging Your Application*, or Appendix B, *Error Codes and Solutions.*

Count Variables – ibcnt and ibcntl

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

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

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

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

Using IBIC 488.2 to Communicate with Devices

Before you begin writing your application program, you might want to use the Interface Bus Interactive Control utility, `IBIC 488.2`. With `IBIC 488.2`, you communicate with your instruments from the keyboard rather than from an application program. Before you develop your GPIB application, you can use `IBIC 488.2` to learn how to communicate with your instruments and to determine your programming needs. For specific device communication instructions, refer to the user manual that came with your instrument. For information about using `IBIC 488.2` and for detailed examples, refer to Chapter 4, *Interface Bus Interactive Control Utility*.

Writing Your NI-488 Application

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

Items to Include

- For C applications, include the GPIB header files `decl.h` and `DrInterface.h`. These file contain variable and constant declarations as well as declarations of structures.
- For QuickBASIC applications, the file `QB488Init.bas` must be merged at the beginning of your program and the library file `QuickBASIC4882.lib` must be present in your `System Folder`.
- Check for errors after each NI-488 function call.
- Declare and define a function to handle GPIB errors. This function takes the device offline and closes the application. If the function is declared as follows:

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

  then your application invokes the function as follows:

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

NI-488 Program Shell

Figure 2-1 is a flowchart of the steps to create your application program using device-level NI-488 functions.

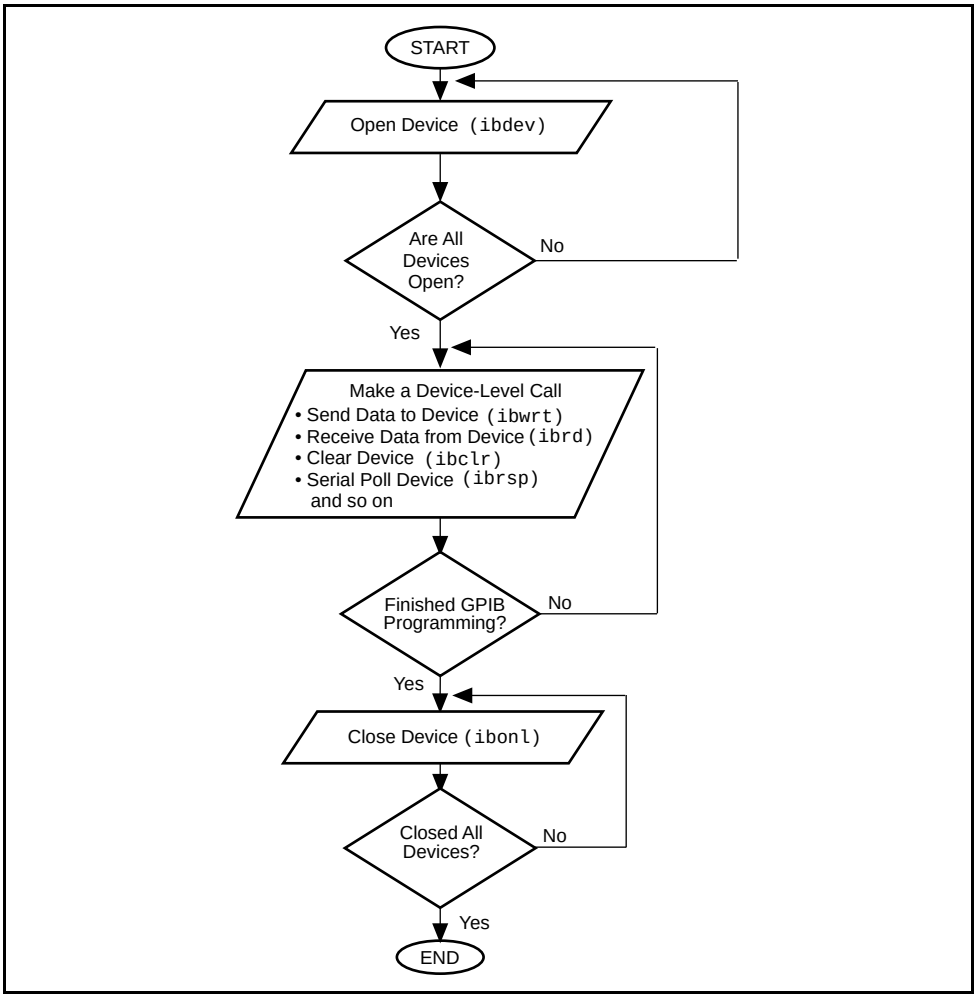


Figure 2-1. General Program Shell Using NI-488 Device Functions

General Program Steps and Examples

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

Step 1. Open a Device

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

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

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

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

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

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

Step 2. Clear the Device

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

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

Step 3. Configure the Device

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

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

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

Step 4. Trigger the Device

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

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

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

Step 5. Wait for the Measurement

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

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

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

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

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

Step 6. Read the Measurement

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

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

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

sum += AsciiToFloat(rdbuf);
```

Step 7. Process the Data

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

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

Step 8. Place the Device Offline

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

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

Writing Your NI-488.2 Application

This section discusses items you should include in an application program that uses NI-488.2 routines, general program steps, and an NI-488.2 example. In this manual the example code is presented in C using the standard C language interface. The NI-488.2 software includes the source code for an example NI-488.2 application written in QuickBASIC (`QBSAMP4882.bas`).

Items to Include

- For C applications, include the GPIB header files `decl.h` and `DrInterface.h`. These file contain variable and constant declarations as well as declarations of structures.
- For QuickBASIC applications, the file `QB488Init.bas` must be merged at the beginning of your program and the library file `QuickBASIC4882.lib` must be loaded by your program.
- Check for errors after each NI-488.2 routine.
- Declare and define a function to handle GPIB errors. This function takes the device offline and closes the application. If the function is declared as follows:

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

  then your application invokes the function as follows:

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

NI-488.2 Program Shell

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

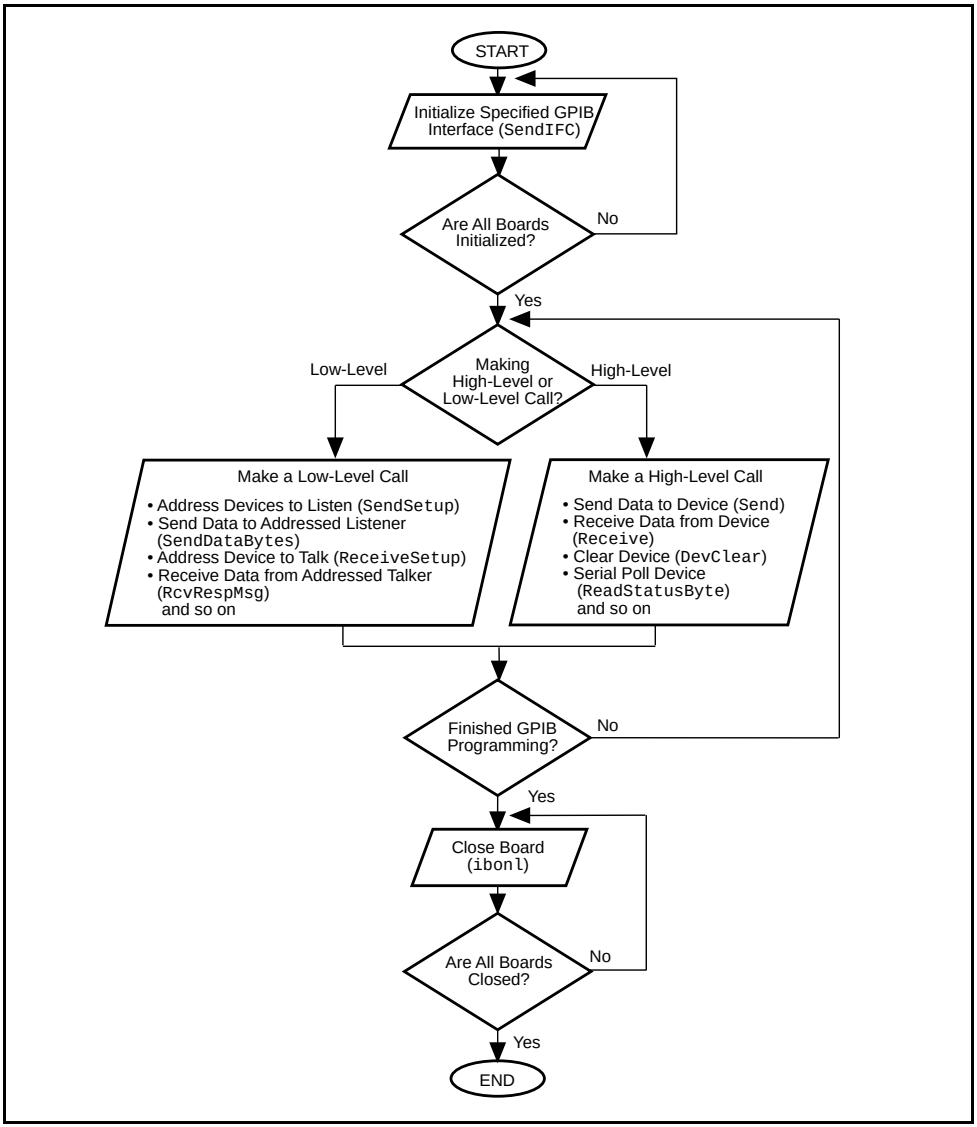


Figure 2-2. General Program Shell Using NI-488.2 Routines

General Program Steps and Examples

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

Step 1. Initialization

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

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

Step 2. Find All Listeners

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

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

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

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

Step 3. Identify the Instrument

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

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

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

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

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

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

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

Step 4. Initialize the Instrument

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

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

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

Step 5. Configure the Instrument

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

The bytes in this example instruct the meter to measure volts alternating current (`VAC`) using auto-ranging (`AUTO`), to wait for a trigger from the Controller before starting a measurement (`TRIGGER 2`), and to assert SRQ when the measurement has been completed and the meter is ready to send the result (`*SRE 16`). The fourth argument represents the number of bytes to be sent. The last argument, `NLend`, is a constant defined in the header file which tells `Send` to append a linefeed character, with EOI asserted, to the end of the message sent to the multimeter.

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

Step 6. Trigger the Instrument

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

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

Step 7. Wait for the Measurement

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

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

After you have detected SRQ, use the `ReadStatusByte` routine to poll the meter and determine its status. The first argument is the GPIB board number, the second argument

is the GPIB address of the instrument, and the last argument is a variable that `ReadStatusByte` uses to store the status byte of the instrument.

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

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

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

Step 8. Read the Measurement

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

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

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

Step 9. Process the Data

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

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

Step 10. Place the Board Offline

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

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

Compiling, Linking, and Running

C Applications

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

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

The file `decl.h` defines external variables and constants that you can use in your application.

If your application requires prototypes, be sure to include the following statement at the beginning of your application program.

```
#define PROTOTYPES
```

The GPIB status, error, and count information are returned in the variables `ibsta`, `iberr`, and `ibcnt`, as described earlier in this chapter.

The file `LI.c` is the language interface source code. A small amount of code is conditionally compiled. Depending on whether you are using THINK C , MPW C, or Metrowerks CodeWarrior C, complete one of the following tasks.

- For THINK C, add the file `LI.c` to your project and add the THINK C library that contains string functions to your project.
- For MPW C, compile the file `LI.c` with the following command to produce the object module called `LI.c.o`.

  ```
  c -d MPW LI.c
  ```

  Add `LI.c.o` to the command that links your object module to create the application.
- For Metrowerks CodeWarrior C, add the file `LI.c` to your project and add the Metrowerks C libraries that support toolbox and string functions to your project.

QuickBASIC Applications

You must execute a few lines of initialization before the main body of your application program.

Place the file `QB488Init.bas`, which contains initialization statements, at the beginning of the application program. Refer to the Microsoft QuickBASIC MERGE command or use the editor to copy and paste. `QB488Init.bas` contains code that loads the library, `QuickBASIC4882.lib`, and initializes three NI-488 status variables. This

library contains the Microsoft QuickBASIC language interface to the NI-488.2 driver. Place the library in the `System Folder`.

The GPIB status, error, and count information are returned in the variables `ibsta%`, `iberr%`, and `ibcnt&`, as described earlier in this chapter.

A library routine cannot pass a value back to QuickBASIC through either a temporary or an uninitialized variable. You cannot use temporary variables to receive data from any NI-488 function or NI-488.2 routine. Refer to the *Microsoft QuickBASIC* manual for examples of temporary variables.

Chapter 3
Debugging Your Application

This chapter describes several ways to debug your application program.

Running NI-488.2 Test

The software diagnostic test `NI-488.2 Test` verifies that the NI-488.2 software is installed and functioning with the GPIB board. For more information about `NI-488.2 Test`, refer to the getting started manual that came with your GPIB board.

Debugging with the Global Status Variables

After each function call to your NI-488.2 driver, `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are updated before the call returns to your application. You should check for an error after each GPIB call. Refer to Chapter 2, *Developing Your Application*, for more information about how to use these variables within your program to automatically check for errors.

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

Debugging with IBIC 488.2

If your application does not automatically check for and display errors, you can locate an error by using the Interface Bus Interactive Control utility, `IBIC 488.2`. Simply issue the same functions or routines, one at a time as they appear in your application program. Because `IBIC 488.2` returns the status values and error codes after each call, you should be able to determine which GPIB call is failing. For more information about `IBIC 488.2`, refer to Chapter 4, *Interface Bus Interactive Control Utility*.

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

GPIB Error Codes

Table 3-1 lists the GPIB error codes. Remember that the error variable is meaningful only when the ERR bit in the status variable is set. For a detailed description of each error and possible solutions, refer to Appendix B, *Error Codes and Solutions*.

Table 3-1. GPIB Error Codes

Error Mnemonic	iberr Value	Meaning
EDVR	0	System error
ECIC	1	Function requires GPIB board to be CIC
ENOL	2	No Listeners on the GPIB
EADR	3	GPIB board not addressed correctly
EARG	4	Invalid argument to function call
ESAC	5	GPIB board not System Controller as required
EABO	6	I/O operation aborted (timeout)
ENEB	7	Nonexistent GPIB board
EDMA	8	No DMA channel available
EOIP	10	Asynchronous I/O in progress
ECAP	11	No capability for operation
EFSO	12	File system error
EBUS	14	GPIB bus error
ESTB	15	Serial poll status byte queue overflow
ESRQ	16	SRQ stuck in ON position
ETAB	20	Table problem
ELCK	21	Board or device is locked

Configuration Errors

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

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

If these problems occur, make sure that the GPIB hardware settings match the NI-488.2 software settings for the interrupt request level and the DMA channel. Refer to the getting started manual that came with your kit for information on hardware and software default settings. For instructions on how to view or modify the NI-488.2 software configuration, refer to Chapter 6, *GPIB Configuration Utility*.

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

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

Timing Errors

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

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

Communication Errors

Repeat Addressing

Some devices require GPIB addressing before any GPIB activity. Devices adhering to the IEEE 488.2 standard should remain in their current state until specific commands are sent across the GPIB to change their state. You might need to configure your NI-488.2 driver to perform repeat addressing if your device does not remain in its currently addressed state. Refer to Chapter 6, *GPIB Configuration Utility*, or to the description of `ibconfig` (option `IbcREADDR`) in the *NI-488.2 Function Reference Manual for Macintosh* for more information about reconfiguring your software.

Termination Method

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

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

Common Questions

What do I do if `NI-488.2 Test` fails with an error?

Refer to the getting started manual for specific information about what might cause this test to fail.

How do I communicate with my instrument over the GPIB?

Refer to the documentation that came from the instrument manufacturer. The command sequences you use are totally dependent on the specific instrument. The documentation for each instrument should include the GPIB commands you need to communicate with it. In most cases, NI-488 device-level calls are sufficient for communicating with instruments. Refer to Chapter 2, *Developing Your Application*, for more information.

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

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

What do I do if I have installed the NI-488.2 software and now my Macintosh crashes upon startup?

Try changing the name of the `NI-488 INIT` to `ZNI-488 INIT`. Because INITs load in alphabetical order, the `ZNI-488 INIT` will load last, preventing possible corruption from INITs that load after it. If changing the name of the `NI-488 INIT` does not solve the problem, another INIT file might have a conflict with the `NI-488 INIT`. Try removing some of your other INIT files. You can store them in a temporary folder, in case you need to reload them later. If you are using System 7.5 or later, you can use the `Extensions Manager` control panel to disable certain extensions and control panels.

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

Examine the value of `ibsta` after each NI-488 or NI-488.2 call. If a call fails, the ERR bit of `ibsta` is set and an error code is stored in `ibcnt`. For more information about global status variables, refer to Chapter 2, *Developing Your Application.*

How can I use the files located in the `Ethernet` folder?

You do not need to use the files in the `Ethernet` folder unless you have a National Instruments GPIB-ENET.

How do I use `IBIC 488.2`?

You can use `IBIC 488.2` to practice communication with your instrument, troubleshoot problems, and develop your application program. For instructions, refer to Chapter 4, *Interface Bus Interactive Control Utility.*

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

Run the `NI-Boards` configuration utility for information about the GPIB boards installed in your computer.

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

Select the `NI-488 INIT` by clicking on it once, and type <Command-I> to get version information.

What information should I have before I call National Instruments?

Before you contact National Instruments, note the results of the diagnostic test `NI-488.2 Test` and fill out the support forms in Appendix D, *Customer Communication*.

Chapter 4
Interface Bus Interactive Control Utility

This chapter introduces you to IBIC 488.2, the interactive control utility you can use to communicate with GPIB devices interactively.

Overview

With the IBIC 488.2 utility, you communicate with GPIB devices through functions you enter at the keyboard. For specific information about how to communicate with your particular device, refer to the manual that came with the device. You can use IBIC 488.2 to practice communication with the instrument, troubleshoot problems, and develop your application program.

One way IBIC 488.2 helps you to learn about your instrument and to troubleshoot problems is by displaying the following information on your screen whenever you enter a command:

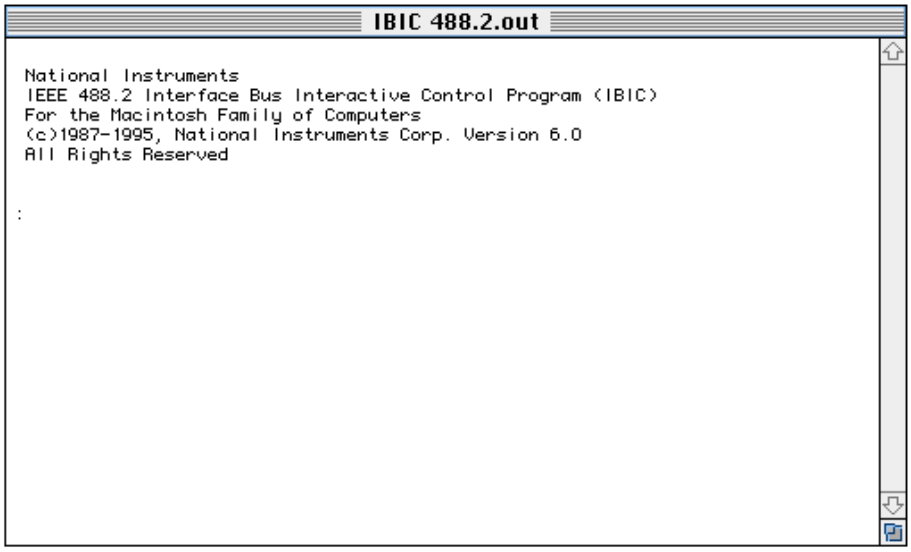
- The results of the status word (ibsta) in hexadecimal notation
- The mnemonic constant of each bit set in ibsta
- The mnemonic value of the error variable (iberr) if an error exists (the ERR bit is set in ibsta)
- The count value for each read, write, or command function
- The data received from your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

Example Using NI-488 Functions

This section shows how you might use IBIC 488.2 to test a sequence of NI-488 device function calls. You do not need to remember the parameters that each function takes. If you enter the function name only, IBIC 488.2 prompts you for the necessary parameters.

1. Run IBIC 488.2 by double-clicking on the IBIC 488.2 icon.

Your screen should appear as follows:



2. Use `ibdev` to open a device, assign it to access board `gpib0`, choose a primary address of 6 with no secondary address, set a timeout of 10 s, enable the END message, and disable the EOS mode:

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

ud0:
```

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

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

ud0:
```

3. Clear the device as follows:

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

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

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

or

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

5. Trigger the device as follows:

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

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

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

or

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

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

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

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

```
ud0: ibrd
   enter byte count: 18
[0100] (cmpl)
count: 18
4e 44 43 56 20 30 30 30     N D C V   0 0 0
2e 30 30 34 37 45 2b 30     . 0 0 4 7 E + 0
0a 0a                       . .
```

or

```
ud0: ibrd 18
[0100] (cmpl)
count: 18
4e 44 43 56 20 30 30 30     N D C V   0 0 0
2e 30 30 34 37 45 2b 30     . 0 0 4 7 E + 0
0a 0a                       . .
```

9. Place the device offline as follows:

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

or

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

10. Terminate the `IBIC 488.2` program by entering `q` at the prompt or choosing **Quit** from the **File** menu.

IBIC 488.2 Syntax

When you enter commands in `IBIC 488.2`, you can either include the parameters, or the program prompts you for values. Some commands require numbers as input values. Others might require you to input a string.

Number Syntax

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

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

Octal numbers–You must precede octal numbers by zero only (for example, 015).

Decimal numbers–Enter the number only.

String Syntax

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

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

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

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

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

Address Syntax

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

IBIC 488.2 Syntax for NI-488 Functions

Table 4-1 and Table 4-2 summarize the syntax of NI-488 functions in `IBIC 488.2`. `v` represents a number and `string` represents a string that you input. For more information about the function parameters, use the `IBIC 488.2` help feature or refer to the *NI-488.2 Function Reference Manual for Macintosh*.

Table 4-1. Syntax for Device-Level NI-488 Functions in IBIC 488.2

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

Table 4-2. Syntax for Board-Level NI-488 Functions in IBIC 488.2

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

IBIC 488.2 Syntax for NI-488.2 Routines

Table 4-3 summarizes the syntax of NI-488.2 routines in `IBIC 488.2. v` represents a number and `string` represents a string. `address` represents an address, and `addrlist` represents a list of addresses separated by commas. For more information about the routine parameters, use the `IBIC 488.2` help feature or refer to the *NI-488.2 Function Reference Manual for Macintosh.*

Table 4-3. Syntax for NI-488.2 Routines in IBIC 488.2

Routine Syntax	Description
`AllSpoll addrlist`	Serial poll multiple devices
`DevClear address`	Clear a device
`DevClearList addrlist`	Clear multiple devices
`EnableLocal addrlist`	Enable local control
`EnableRemote addrlist`	Enable remote control
`FindLstn addrlist limit`	Find all Listeners
`FindRQS addrlist`	Find device asserting SRQ
`PassControl address`	Pass control to a device
`PPoll`	Parallel poll devices
`PPollConfig address line sense`	Configure device for parallel poll
`PPollUnconfig address`	Unconfigure device for parallel poll
`RcvRespMsg address string mode`	Receive response message
`ReadStatusByte address`	Serial poll a device
`Receive address string mode`	Receive data from a device
`ReceiveSetup address`	Receive setup
`ResetSys addrlist`	Reset multiple devices
`Send address string mode`	Send data to a device
`SendCmds string`	Send command bytes
`SendDataBytes addrlist string mode`	Send data bytes
`SendIFC`	Send interface clear
`SendList addrlist string mode`	Send data to multiple devices
`SendLLO`	Put devices in local lockout
`SendSetup addrlist`	Send setup
`SetRWLS addrlist`	Put devices in remote with lockout state
`TestSys addrlist`	Cause multiple devices to perform self tests
`TestSRQ`	Test for service request
`Trigger address`	Trigger a device
`TriggerList addrlist`	Trigger multiple devices
`WaitSRQ`	Wait for service request

Status Word

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

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

ud0:
```

For more information about the status word, refer to Chapter 2, *Developing Your Application*.

Error Information

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

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

ud0:
```

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

For a detailed list of the error codes and their meanings, refer to Chapter 3, *Debugging Your Application*.

Count

When an I/O function completes, `IBIC 488.2` displays the actual number of bytes sent or received, regardless of the existence of an error condition.

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

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

Common NI-488 Functions

ibfind

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

```
:ibfind gpib0
id = 32000

gpib0:
```

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

Any name you use with the `ibfind` function must be a valid symbolic name in the driver. For more information about valid names, refer to Chapter 6, *GPIB Configuration Utility*.

ibdev

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

With `ibdev`, you specify the following values:

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

The following example shows `ibdev` opening an available device and assigning it to access `gpib0` (`board` = 0) with a primary address of 6 (`pad` = 6), a secondary address of hex 67 (`sad` = 0x67), a timeout of 10 s (`tmo`=13), the END message enabled (`eot` =1), and the EOS mode disabled (`eos` = 0).

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

ud0:
```

If you use `ibdev` without specifying parameters, `IBIC 488.2` prompts you for the input parameters as shown in the following example:

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

ud0:
```

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

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

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

  :
  ```

- ENEB – The board index entered refers to a known board (such as 0), but the driver cannot find the board.

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

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

  ud0: ibpad 6
  previous value: 16
  ```

ibwrt

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

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

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

ibrd

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

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

Common NI-488.2 Routines in IBIC 488.2

Set

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

`set 488.2` *n*

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

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

```
set 488.2 0

488.2 (0):
```

Send and SendList

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

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

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

Receive

The `Receive` routine causes the GPIB board to receive data from another GPIB device. The following example acquires 10 data bytes from the device at primary address 5. It stops receiving data when 10 characters have been received or when the END message is received. The acquired data is then displayed in hex format along with its ASCII equivalent. The `IBIC 488.2` program also displays the status word and the count of transferred bytes.

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

Auxiliary Functions

Table 4-4 summarizes the auxiliary functions that you can use in `IBIC 488.2`.

Table 4-4. Auxiliary Functions in IBIC 488.2

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

Set (Select Device or Board)

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

```
488.2 (0): set ud0

ud0:
```

Help (Display Help Information)

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

! (Repeat Previous Function)

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

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

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

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

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

The `+` function turns the screen output on.

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

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

ud0: -

ud0: ibrd 8

ud0: +

ud0: ibrd 8
[2100] (end cmpl)
count: 8
71 72 73 74 75 76 77 78    q r s t u v w x
```

n* (Repeat Function n Times)

The `n*` function repeats the execution of the specified function `n` times, where `n` is an integer. In the following example, the message `Hello` is sent five times to the device described by `ud0`.

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

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

```
ud0: 5*ibwrt "Hello"
ud0: 20* !
ud0: 10* !
```

Notice that the multiplier (`*`) does not become part of the function name; that is, `ibwrt "Hello"` is repeated 20 times, not `5* ibwrt "Hello"`.

$ (Execute Indirect File)

The `$` function reads a specified file and executes the `IBIC 488.2` functions listed in that file, in sequence, as if they were entered in that order from the keyboard. The following example executes the `IBIC 488.2` functions listed in the file `userfile`.

```
gpib0: $ userfile
```

The following example repeats the operation three times.

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

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

Print (Display the ASCII String)

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

```
dev1: print "hello"
hello

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

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

Buffer (Set Buffer Display Mode)

You can set the type of display used for buffers to control how much of the buffer is displayed. Type `buffer 0` to turn off the display of all buffers, `buffer 1` to display the buffer in ASCII only, `buffer 2` to display the buffer in hex/ASCII, or `buffer 3` to display a brief hex/ASCII display.

Chapter 5
GPIB Programming Techniques

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

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

Termination of Data Transfers

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

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

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

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

You can use the `ibconfig` function to configure the software to inform you whether or not the GPIB EOI line was asserted when the EOS byte was read. Use the `IbcEndBitIsNormal` option to configure the software to report only the END bit in

`ibsta` when the GPIB EOI line is asserted. By default, the NI-488.2 driver reports END in `ibsta` when either the EOS byte is read in or the EOI line is asserted during a read.

High-Speed Data Transfers (HS488)

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

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

Enabling HS488

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

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

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

If you are using board-level functions or NI-488.2 routines and you want to configure devices for high-speed, you must send the HS488 configuration messages using `ibcmd` or `SendCmds`. The HS488 configuration message is made up of two GPIB command bytes. The first byte, the Configure Enable (CFE) message (hex 1F), places all HS488 devices into their configuration mode. Non-HS488 devices should ignore this message. The second byte is a GPIB secondary command that indicates the number of meters of cable in your system. It is called the Configure (CFGn) message. Because HS488 can operate only with cable lengths of 1 to 15 meters, only CFGn values of 1 through 15 (hex 61 through 6F) are valid. If the cable length was configured correctly in the

configuration utility, you can determine how many meters of cable are in your system by calling `ibask` (option `IbaHSCableLength`) in your application program. For CFE and CFGn messages, refer to Appendix A, *Multiline Interface Messages*, in the *NI-488.2 Function Reference Manual for Macintosh.*

System Configuration Effects on HS488

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

Waiting for GPIB Conditions

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

Device-Level Calls and Bus Management

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

- If your GPIB board is configured as the System Controller (default), it automatically makes itself the CIC by asserting the IFC line the first time you make a device-level call.

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

  1. The GPIB interface board asserts the SRQ line.

2. The current CIC serial polls the board.

3. The interface board returns a response byte of hex 42.

4. The current CIC passes control to the GPIB board.

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

Talker/Listener Applications

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

Waiting for Messages from the Controller

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

Requesting Service

Another type of event that might be important in a Talker/Listener application is the serial poll. A Talker/Listener application can call `ibrsv` with a serial poll response byte when it needs to request service from the Controller.

Serial Polling

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

Service Requests from IEEE 488 Devices

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

Service Requests from IEEE 488.2 Devices

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

Automatic Serial Polling

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

1. To enable autopolling, use the configuration utility or the configuration function, `ibconfig` with option `IbcAUTOPOLL`. (By default, autopolling is enabled.)
2. When the SRQ line is asserted, the driver automatically serial polls the open devices.
3. Each positive serial poll response (bit 6 or hex 40 is set) is stored in a queue associated with the device that sent it. The RQS bit of the device status word, `ibsta`, is set.
4. The polling continues until SRQ is unasserted or an error condition is detected.

5. To empty the queue, use the `ibrsp` function. `ibrsp` returns the first queued response. Other responses are read in first-in-first-out (FIFO) fashion. If the RQS bit of the status word is not set when `ibrsp` is called, a serial poll is conducted and returns whatever response is received. You should empty the queue as soon as an automatic serial poll occurs, because responses might be discarded if the queue is full.

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

Stuck SRQ State

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

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

Autopolling and Interrupts

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

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

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

C "ON SRQ" Capability

C applications can respond asynchronously to SRQ using the NI-488 `ibsrq` function. This function lets an application specify an SRQ-handling routine that is called whenever the NI-488.2 driver detects that the SRQ line is asserted. This SRQ-handling routine is

not an interrupt service routine. The driver checks the GPIB SRQ line after any NI-488 function or NI-488.2 routine has completed, and if SRQ is asserted and the application has called `ibsrq`, the user-defined SRQ-handling routine is called.

SRQ and Serial Polling with NI-488 Device Functions

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

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

```
#include "decl.h"

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

       ibwait ( DeviceHandle, TIMO | RQS );

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

SRQ and Serial Polling with NI-488.2 Routines

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

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

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

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

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

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

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

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

Example 1: Using FindRQS

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

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

    WaitSRQ (0, &WaitResult);

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

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

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

  return;
}
```

Example 2: Using AllSpoll

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

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

   WaitSRQ (0, &WaitResult);

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

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

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

   return;
}
```

Parallel Polling

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

Implementing a Parallel Poll

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

Parallel Polling with NI-488 Functions

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

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

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

   0 1 1 E S D2 D1 D0

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

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

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

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

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

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

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

/* Pass the binary bit pattern, 0110 110 or hex 66, to ibppc.
*/

ibppc(dev, 0x66);
```

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

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

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

2. Conduct the parallel poll using `ibrpp` and check the response for a certain value. The following example code performs the parallel poll and compares the response to hex 10, which corresponds to DIO5. If that bit is set, the `ist` of the device is 0.

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

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

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

Parallel Polling with NI-488.2 Routines

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

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

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

/* The following command clears the GPIB.   */

SendIFC(0);

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

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

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

```
PPoll(0, &response);

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

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

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

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

Chapter 6
GPIB Configuration Utility

This chapter contains instructions for configuring the NI-488.2 software with the NI-488 Config utility.

Overview

You can use the GPIB configuration utility, NI-488 Config, to view or change the configuration settings of your NI-488.2 software. With NI-488 Config, you can change the default GPIB settings that your interface board uses to communicate with other devices. The utility edits the default GPIB configuration resources in the NI-488 INIT file. Help is available on the screen for modifying the current settings.

For specific information about possible settings, refer to the getting started manual that came with your GPIB interface board or box.

Running the Configuration Utility

This section contains information on running the NI-488 Config configuration utility. It explains how to use the utility and describes the configuration settings that you can modify.

Opening the Configuration Utility

The NI-488 Config configuration utility appears in the Control Panels folder when you install your NI-488.2 software. Open the Control Panels folder by choosing **Control Panels** from the **Apple** menu .

To access NI-488 Config, double-click on the NI-488 Config icon. The utility displays the currently defined values for characteristics of a particular device or bus, such as addressing and timeout information. Help for modifying the current settings is available at the bottom of the window.

The NI-488 Config configuration utility consists of three frames, arranged vertically and separated by a heavy line. Each frame is labeled in Figure 6-1.

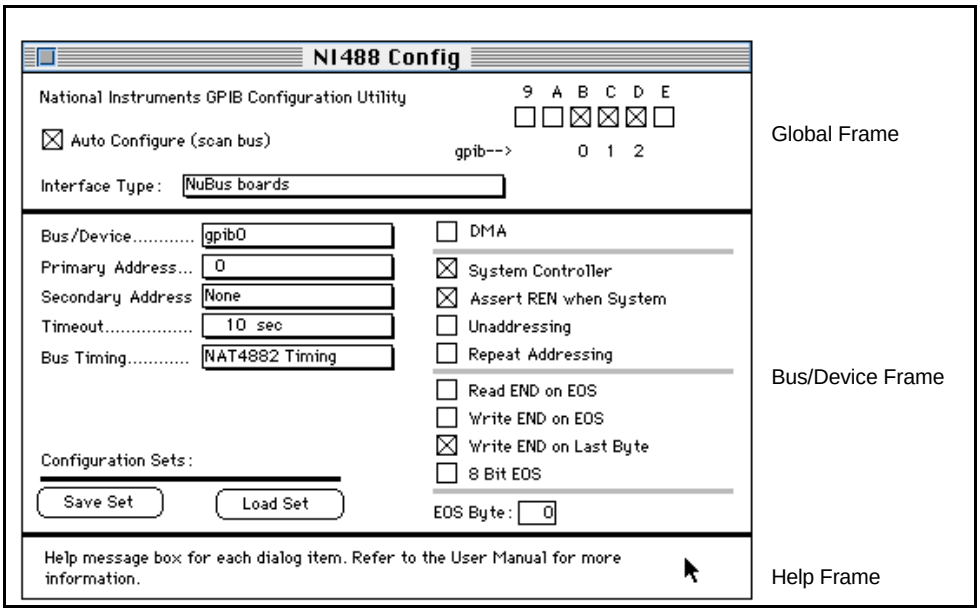


Figure 6-1. Opening Screen of NI-488 Config

The global and bus/device frames contain the configuration characteristic settings. The help frame displays information about the item over which the cursor is positioned.

The **Interface Type** and **Bus/Device** menus affect the display of configuration controls. For example, selecting a serial interface hides the **Auto Configure** checkbox.

Default Configuration

Your NI-488.2 software is shipped with the following default configurations:

- The **Auto Configure** checkbox is selected.
- All buses are configured as shown in the bus/device frame in Figure 6-1.
- All devices are configured similarly to `dev1` shown in the bus/device frame in Figure 6-2. The devices `dev1` through `dev30` use bus `gpib0` and are at the primary addresses 1 through 30, respectively. The devices `dev31` through `dev60` use bus `gpib1` and are at the primary addresses 1 through 30, respectively. The devices `dev61` through `dev64` use bus `gpib2` and are at the primary addresses 1 through 4, respectively.

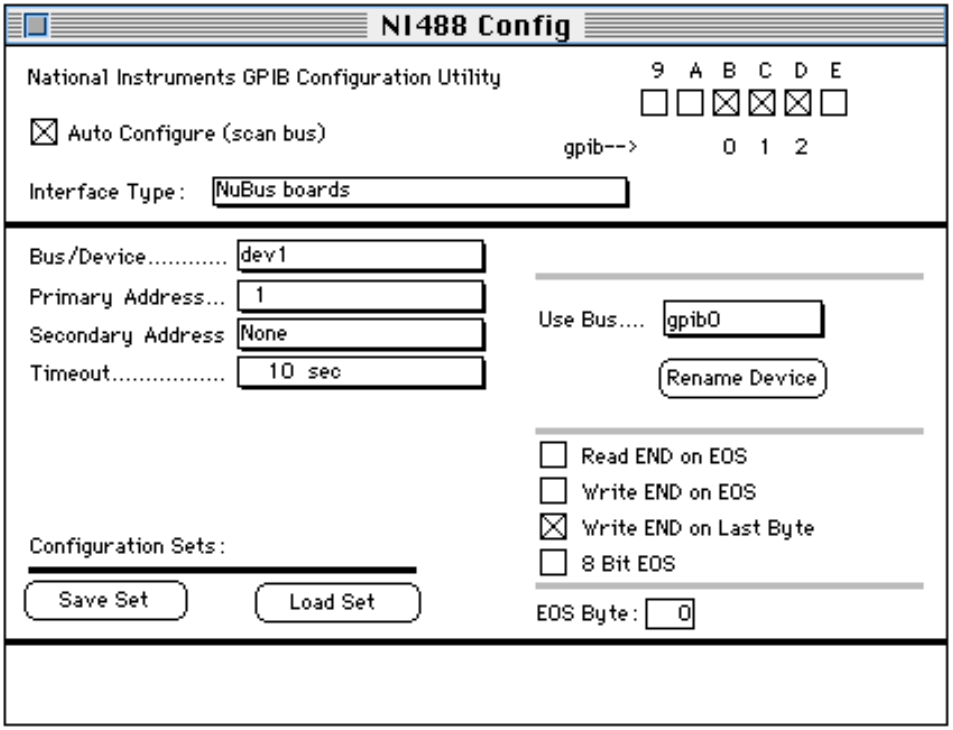
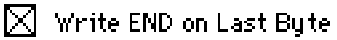


Figure 6-2. Device Default Settings in NI-488 Config

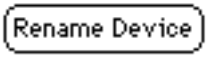
Control Items

`NI-488 Config` has four types of control items:

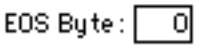
The rectangular boxes with drop shadows and labels to the left have pop-up menus of options. The currently selected option is displayed in the box. To select an option on the pop-up menu, click and hold down the mouse button when the cursor is over the box.



A checkbox is a small square box that contains an X when selected and is labeled at the right or on the top. An unselected checkbox displays an alert box when clicked.



A button is a rounded rectangular box.



An editable text box is a rectangular box labeled to the left.

Help Frame

When you place the cursor over any configuration item, a help message for that item appears in the help frame. Figure 6-3 shows the default configuration for bus `gpib0`. The global frame shows the automatic association of bus `gpib0` with a GPIB board installed in System slot 3 (NuBus slot xB). The cursor is positioned over the **Auto Configure** checkbox and a corresponding help message appears in the help frame.

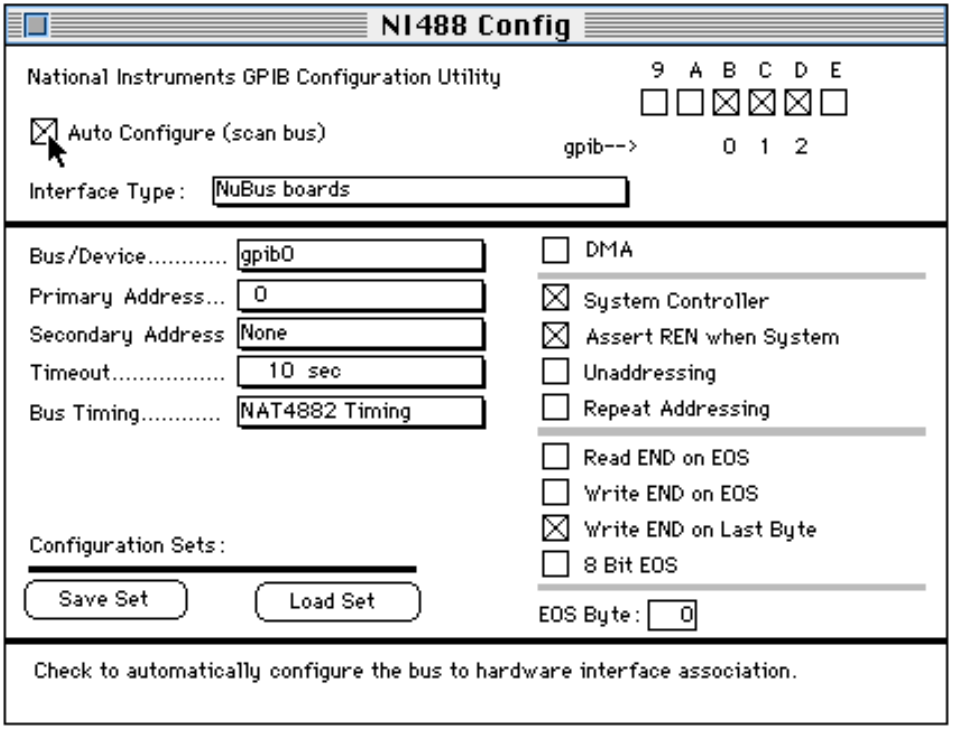


Figure 6-3. Help Frame in NI-488 Config

Global Frame

The **Interface Type** pop-up menu options let you switch the checkboxes among interface types. Choose **NuBus boards** for boards, such as the NB-GPIB-P/TNT or NB-GPIB-P, installed in a NuBus Macintosh. The **Serial box products** option applies to the GPIB-422CT or GPIB-232CT-A, the **Ethernet box products** option applies to the GPIB-ENET, and the **SCSI box products** option applies to the GPIB-SCSI or GPIB-SCSI-A. For specific information on configuring one of those products, refer to the getting started manual that came with the product.

To the upper right of the **Interface Type** menu box is a row of interface checkboxes with which you can associate an IEEE 488 bus. Slot numbers appear above the checkboxes, and associated bus numbers, if any, appear below the checkboxes. To manually associate a bus with an interface, first unselect **Auto Configure**. When you select an interface checkbox with **Auto Configure** selected, the next available bus is assigned to it. Figure 6-4 shows the manual association of bus 0 to System slot 3 (NuBus slot xB).

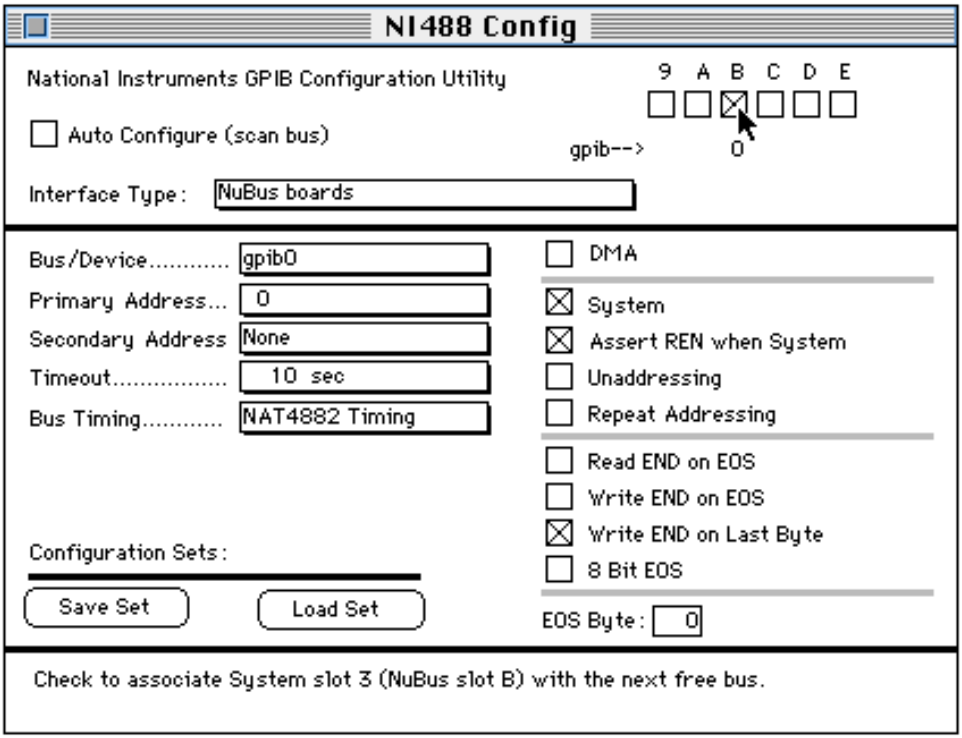


Figure 6-4. Manual Bus Association in NI-488 Config

Selecting the **Auto Configure** checkbox in the global frame automatically configures the buses according to the way the boards are contained in the system. When **Auto Configure** is checked, each bus, `gpib0` through `gpib7`, is associated with the next GPIB board found by the Slot Manager when searching System slots 1 through 6 (NuBus slots 9 through E) and expansion slots x1 through x8 (NuBus slots 1 through 8). Select the **Auto Configure** checkbox unless your application requires compatibility with older releases of the NI-488.2 driver, where the naming conventions of the buses are different. Do *not* check the **Auto Configure** checkbox if you want to change the order that device-identifying software uses GPIB interfaces.

Bus/Device Frame

Items in the bus/device frame configure characteristics of a bus, a device, or either. Table 6-1 lists the primary bus/device options available in `NI-488 Config`. The sections following the table describe the options in more detail.

For information on product-specific options, such as the **Serial** or **IP Address** pop-up boxes, refer to the getting started manual that came with your GPIB hardware.

Table 6-1. Bus/Device Options in NI-488 Config

Option	Type	Default Setting
Primary Address	Bus/Device	0
Secondary Address	Bus/Device	None
Timeout	Bus/Device	10 sec
Read END on EOS	Bus/Device	Disabled
Write END on EOS	Bus/Device	Disabled
Write END on Last Byte	Bus/Device	Enabled
8-bit EOS	Bus/Device	Disabled
EOS Byte	Bus/Device	0
Bus Timing	Bus Only	(Interface-specific)
TNT High Speed	Bus Only	Disabled
DMA	Bus Only	Disabled
System Controller	Bus Only	Enabled
Assert REN when System	Bus Only	Enabled
Unaddressing	Bus Only	Disabled
Repeat Addressing	Bus Only	Disabled
Rename Device	Device Only	dev1-dev64
Use Bus	Device Only	gpib0

Options for Buses or Devices

Select the device or bus you want to configure from the **Bus/Device** pop-up menu. The following sections describe the options available for buses or devices. Also refer to the subsequent sections *Options for Buses Only* and *Options for Devices Only*.

Primary Address

Each device and bus must have unique primary addresses in the range decimal 0 to decimal 30 (hex 1E). The primary GPIB address of any device is set within that device, either with hardware switches or, in some cases, a software program. This address must match the address listed in the configuration utility. Refer to the device documentation for instructions about the device address. The primary GPIB address of all NI-488.2 driver buses is 0, unless changed by the configuration utility. There are no hardware switches on the interface board to select the GPIB address. Use the **Primary Address** pop-up menu to select the primary address of the bus or device.

Secondary Address

You must assign a secondary address in the range decimal 96 (hex 60) to decimal 126 (hex 7E) to any device or bus using secondary addressing. As with primary addressing, the secondary GPIB address of any device is set within that device, either with hardware switches or, in some cases, a software program. This address must match the address listed in the configuration utility. Refer to your device documentation for instructions. By default, secondary addressing is disabled for all devices and boards unless you change it with the configuration utility.

Select the secondary address of the bus or device from the **Secondary Address** pop-up menu. The secondary addresses are displayed in three formats: zero-based, decimal, and hexadecimal. Only the zero-based format is displayed in the pop-up menu box. Selecting **None** means that only primary addressing is used for this bus or device. If you configure any bus or device for secondary addressing, all buses and devices used by the application must be configured for secondary addressing.

Timeout

The timeout value is the approximate length of time that can elapse before I/O functions complete. Select the I/O timeout of the bus or device from the **Timeout** pop-up menu. The abbreviations used in the **Timeout** pop-up menu are: µ**sec** (microseconds), **msec** (milliseconds), and **sec** (seconds). Selecting **None** means I/O for this bus or device will never time out.

EOS Modes

The options described below determine how the device I/O transmissions terminate:

- **Read END on EOS** – Some devices send an EOS byte signaling the last byte of a data message. Checking this box causes the NI-488.2 software to terminate read operations when it receives the EOS byte.
- **Write END on EOS** – Checking this box causes the NI-488.2 software to assert the EOI (send END) line when the EOS character is sent.
- **Write END on Last Byte** – Some devices, as Listeners, require that the Talker terminate a data message by asserting the EOI signal line (sending END) with the last byte. Checking this box causes the NI-488.2 software to assert EOI on the last data byte.
- **8-bit EOS**– Along with the designation of an EOS character, you can specify whether all eight bits are compared to detect EOS, or if just the seven least significant bits (ASCII or ISO format) are compared to detect EOS.

EOS Byte

You can program some devices to terminate a read operation when a selected character is detected. A linefeed character (decimal 10) is a popular EOS character.

Notice that to send the EOS character to a device in a write operation, you must explicitly include that byte in your data string.

Enter the EOS byte (0 to 255) of the bus or device in the **EOS Byte** editable text box. To change the EOS byte, click inside the box, enter the new number, and press the <return> key.

Options for Buses Only

Select the device you want to configure from the **Bus/Device** pop-up menu. The following sections describe the available bus options. See also the section *Options for Buses or Devices* earlier in this chapter.

Bus Timing

This pop-up menu appears when configuring a bus associated with a NAT4882-based interface, such as the NB-GPIB-P. You can use it to specify the T1 delay of the board source handshake capability. This delay determines the minimum interval following Ready for Data (RFD) after which the board may assert Data Valid (DAV) during a write or command operation. If the total length of the GPIB cable in the system is less than

15 m and all devices are *on*, you can choose the sub-item **Very High** (350 ns) from the **NAT4882 Timing** pop-up menu. For total cable lengths greater than 15 m, choose **Low** (2 µs) or **High** (500 ns) depending on the maximum capability of your particular device.

TNT High Speed

If you are using a National Instruments TNT4882C-based interface, such as the NB-GPIB-P/TNT, a second item, **TNT High Speed**, appears enabled. Initially, the sub-item **High Speed Mode Disabled** is checked. If your device is capable of 1-wire high-speed handshaking, you can enable the HS488 high-speed protocol by choosing the sub-item corresponding to the total GPIB cable length of your setup. For maximum performance, select the sub-item **GPIB cable is 1 meter**.

DMA

When the **DMA** box is checked, direct memory access hardware is used for data transfers, freeing the CPU for other work. Uncheck the **DMA** box to transfer data using the CPU. DMA channels are allocated for GPIB when you check the **DMA** box or call the `ibdma` function with v = 1 in an application program.

System Controller

Generally, the NI-488.2 driver is the System Controller (SC). In some situations, such as in a network of computers linked by the GPIB, another device might be System Controller. Selecting the **System Controller** box designates the NI-488.2 driver as System Controller. Unselecting the box designates that it is *not* System Controller. Each bus can have only one System Controller.

Assert REN when System (Controller)

Some devices must be in remote state to communicate over the GPIB. Checking this box permits the driver to assert the Remote Enable condition (REN) when it is System Controller, placing all instruments subsequently addressed into remote state.

Unaddressing

Some devices must be unaddressed after each data or command transfer. To force unaddressing commands to be sent at the end of device functions, check the **Unaddressing** box. (Unchecking the **Unaddressing** box slightly improves the performance of your application, because unaddressing commands are not sent at the end of device functions.)

Repeat Addressing

Normally, a device remains addressed after a read or write operation is performed. However, some devices require addressing for each operation. If you check the **Repeat Addressing** box, read or write operations readdress the selected device even if the same operation was just performed with that device.

Options for Devices Only

Select the device you want to configure from the **Bus/Device** pop-up menu. The device is connected to the bus number that appears in the **Use Bus** text box. The following sections describe the available device options. See also the section *Options for Buses or Devices* earlier in this chapter.

Rename Device

You can rename the device displayed in the **Bus/Device** pop-up menu by clicking the **Rename Device** button and entering the new name. This feature is helpful when configuring a large number of devices, because the new name of the device that you entered appears in the **Bus/Device** pop-up menu. However, to avoid the confusion of naming and renaming devices, use the NI-488 function `ibdev` in new applications to dynamically configure new devices. You can use `ibdev` to configure the driver from your program instead of from the configuration utility.

Use Bus

You can connect the device displayed on the **Bus/Device** pop-up menu to a different bus by selecting the new bus from the **Use Bus** pop-up menu. The new bus number appears to the left of the device name in the **Bus/Device** pop-up menu.

Exiting the Configuration Utility

To exit the configuration utility, click on the close box in the upper left corner of the configuration screen.

An alert message displays if you close the utility while any of the following conditions applies.

- The Macintosh must be restarted to load new drivers or change the serial port settings.
- A device GPIB address conflicts with the GPIB address of the bus to which it is connected. Each GPIB address must be unique.

- No GPIB board is in the slot associated with one of the buses.
- A bus or device I/O timeout is set to **None** (disabled).

Appendix A
Status Word Conditions

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

For information about how to use `ibsta` in your application program, refer to Chapter 2, *Developing Your Application.*

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

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

The following table lists the status word bits.

Mnemonic	Bit Pos.	Hex Value	Type	Description
ERR	15	8000	dev, brd	GPIB error
TIMO	14	4000	dev, brd	Time limit exceeded
END	13	2000	dev, brd	END or EOS detected
SRQI	12	1000	brd	SRQ interrupt received
RQS	11	800	dev	Device requesting service
CMPL	8	100	dev, brd	I/O completed
LOK	7	80	brd	Lockout State
REM	6	40	brd	Remote State
CIC	5	20	brd	Controller-In-Charge
ATN	4	10	brd	Attention is asserted
TACS	3	8	brd	Talker
LACS	2	4	brd	Listener
DTAS	1	2	brd	Device Trigger State
DCAS	0	1	brd	Device Clear State

ERR (dev, brd)

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

TIMO (dev, brd)

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

END (dev, brd)

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

Some applications might need to know the exact I/O read termination mode of a read operation – EOI by itself, the EOS character by itself, or EOI plus the EOS character. You can use the `ibconfig` function (option `IbcEndBitIsNormal`) to enable a mode in which the END bit is set only when EOI is asserted. In this mode, if the I/O operation completes because of the EOS character by itself, END is not set. The application should check the last byte of the received buffer to see if it is the EOS character.

SRQI (brd)

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

RQS (dev)

RQS appears in the status word only after a device-level call and indicates that the device is requesting service. RQS is set whenever bit 6 is asserted in the serial poll status byte of the device. The serial poll that obtains the status byte can be the result of a call to

`ibrsp`, or the poll might be automatic if automatic serial polling is enabled. Do not issue an `ibwait` on RQS for a device that does not respond to serial polls. RQS is cleared when an `ibrsp` reads the serial poll status byte that caused the RQS.

CMPL (dev, brd)

CMPL indicates the condition of I/O operations. It is set whenever an I/O operation is complete. CMPL is cleared while the I/O operation is in progress.

LOK (brd)

LOK indicates whether the board is in a lockout state. While LOK is set, the `EnableLocal` routine or `ibloc` function is inoperative for that board. LOK is set whenever the GPIB board detects that the Local Lockout (LLO) message has been sent either by the GPIB board or by another Controller. LOK is cleared when the System Controller unasserts the Remote Enable (REN) GPIB line.

REM (brd)

REM indicates whether or not the board is in the remote state. REM is set whenever the Remote Enable (REN) GPIB line is asserted and the GPIB board detects that its listen address has been sent either by the GPIB board or by another Controller. REM is cleared in the following situations:

- When REN becomes unasserted
- When the GPIB board as a Listener detects that the Go to Local (GTL) command has been sent either by the GPIB board or by another Controller
- When the `ibloc` function is called while the LOK bit is cleared in the status word

CIC (brd)

CIC indicates whether the GPIB board is the Controller-In-Charge. CIC is set when the `SendIFC` routine or `ibsic` function is executed either while the GPIB board is System Controller or when another Controller passes control to the GPIB board. CIC is cleared either when the GPIB board detects Interface Clear (IFC) from the System Controller or when the GPIB board passes control to another device.

ATN (brd)

ATN indicates the state of the GPIB Attention (ATN) line. ATN is set whenever the GPIB ATN line is asserted, and it is cleared when the ATN line is unasserted.

TACS (brd)

TACS indicates whether the GPIB board is addressed as a Talker. TACS is set whenever the GPIB board detects that its talk address (and secondary address, if enabled) has been sent either by the GPIB board itself or by another Controller. TACS is cleared whenever the GPIB board detects the Untalk (UNT) command, its own listen address, a talk address other than its own talk address, or Interface Clear (IFC).

LACS (brd)

LACS indicates whether the GPIB board is addressed as a Listener. LACS is set whenever the GPIB board detects that its listen address (and secondary address, if enabled) has been sent either by the GPIB board itself or by another Controller. LACS is also set whenever the GPIB board shadow handshakes as a result of the `ibgts` function.

LACS is cleared whenever the GPIB board detects the Unlisten (UNL) command, its own talk address, Interface Clear (IFC), or that the `ibgts` function has been called without shadow handshake.

DTAS (brd)

DTAS indicates whether the GPIB board has detected a device trigger command. DTAS is set whenever the GPIB board, as a Listener, detects that the Group Execute Trigger (GET) command has been sent by another Controller. DTAS is cleared on any call immediately following an `ibwait` call, if the DTAS bit is set in the `ibwait` mask parameter.

DCAS (brd)

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

Appendix B
Error Codes and Solutions

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

The following table lists the GPIB error codes.

Error Mnemonic	`iberr` Value	Meaning
EDVR	0	System error
ECIC	1	Function requires GPIB board to be CIC
ENOL	2	No Listeners on the GPIB
EADR	3	GPIB board not addressed correctly
EARG	4	Invalid argument to function call
ESAC	5	GPIB board not System Controller as required
EABO	6	I/O operation aborted (timeout)
ENEB	7	Nonexistent GPIB board
EDMA	8	No DMA channel available
EOIP	10	Asynchronous I/O in progress
ECAP	11	No capability for operation
EFSO	12	File system error
EBUS	14	GPIB bus error
ESTB	15	Serial poll status byte queue overflow
ESRQ	16	SRQ stuck in ON position
ETAB	20	Table problem
ELCK	21	Board or device is locked

EDVR (0)

EDVR is returned when the board or device name passed to `ibfind` is not configured in the software.

EDVR is also returned when an invalid unit descriptor is passed to any function call.

EDVR is also returned when the driver is not installed. In this case, `ibcnt` contains a system level error code.

Solutions

- Use `ibdev` to open a device without specifying its symbolic name.
- Use only device or board names that are configured in the utility program `NI-488 Config` as parameters in the `ibfind` function.
- Use the unit descriptor returned from the `ibfind` function as the first parameter in subsequent NI-488 functions. Examine the variable after the `ibfind` and before the failing function to make sure it was not corrupted.
- Make sure the NI-488.2 driver is installed by checking to see if `NI-488 INIT` is in the `Extensions` folder in the `System Folder`.

ECIC (1)

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

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

Solutions

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

ENOL (2)

ENOL usually occurs when a write operation is attempted with no Listeners addressed. For a device write, this error indicates that the GPIB address configured for that device in the software does not match the GPIB address of any device connected to the bus, that the GPIB cable is not connected to the device, or that the device is not powered on.

ENOL can occur in situations in which the GPIB board is not the CIC and the Controller asserts ATN before the write call in progress has ended.

Solutions

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

EADR (3)

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

EADR is also returned by the function `ibgts` when the shadow-handshake feature is requested and the GPIB ATN line is already unasserted. In this case, the shadow handshake is not possible and the error is returned to notify you of that fact.

Solutions

- Make sure that the GPIB board is addressed correctly before calling `ibrd`, `ibwrt`, `RcvRespMsg`, or `SendDataBytes`.
- Avoid calling `ibgts` except immediately after an `ibcmd` call. (`ibcmd` causes ATN to be asserted.)

EARG (4)

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

- `ibtmo` called with a value not in the range 0 through 17
- `ibpad` or `ibsad` called with invalid addresses
- `ibppc` called with invalid parallel poll configurations
- A board-level NI-488 call made with a valid device descriptor or a device-level NI-488 call made with a board descriptor
- An NI-488.2 routine called with an invalid address
- `PPollConfig` called with an invalid data line or sense bit

Solutions

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

ESAC (5)

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

Solutions

Give the GPIB board System Controller capability by calling `ibrsc 1` or by using `NI-488 Config` to configure that capability into the software.

EABO (6)

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

Frequently, the I/O is not progressing (the Listener is not continuing to handshake or the Talker has stopped talking), or the byte count in the call which timed out was more than the other device was expecting.

Solutions

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

ENEB (7)

ENEB occurs when there is no GPIB board present. This happens when the board is not physically plugged into the system, or there is a conflict in the system.

Solutions

Verify that all GPIB interfaces and external controller boxes are plugged in securely, powered on, and configured properly in the GPIB configuration.

EDMA (8)

EDMA occurs when the driver is unable to allocate a DMA channel.

Solutions

Verify that other boards are not using all seven available DMA channels. Disconnect the RTSI connector from the other DMA boards temporarily.

EOIP (10)

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

Solutions

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

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

ECAP (11)

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

Solutions

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

EFSO (12)

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

Solutions

- Make sure the file is in the same folder as your application.
- Make sure there is enough room on the disk to hold the file.

EBUS (14)

EBUS results when certain GPIB bus errors occur during device functions. All device functions send command bytes to perform addressing and other bus management. Devices are expected to accept these command bytes within the time limit specified by

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

Solutions

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

ESTB (15)

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

Solutions

- Call `ibrsp` more frequently to empty the queue.
- Disable autopolling with the `ibconfig` function or the `NI-488 Config` utility.

ESRQ (16)

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

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

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

Solutions

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

ETAB (20)

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

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

Solutions

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

ELCK (21)

ELCK occurs if the requested GPIB-ENET board or device is being used through another connection.

Solutions

Wait for the lock on the board or device to be released, or try using `ibunlock` if you previously used `iblock` to lock access to the connection.

Appendix C
Device Manager Interface

This chapter contains information for programming your GPIB interface from any language using the Device Manager functions.

The examples in this appendix are in C language syntax.

Overview

You might want to use the Device Manager if you need to make asynchronous calls or you require completion routines for your application. You can make NI-488 calls using the Macintosh Device Manager. The NI-488.2 software also supports the high-level Device Manager routines (`OpenDriver`, `CloseDriver`, and `Control`) and the low-level Device Manager routines (`PBOpen`, `PBClose`, and `PBControl`). Refer to the Device Manager chapter of *Inside Macintosh*, Volume II, for a thorough explanation of these routines.

Opening the GPIB Driver

Before you use any of the Device Manager calls, you must open the NI-488.2 driver with an `OpenDriver` call. This call gives the name of the driver to be opened, `GPIB Driver`, and returns the `refNum` of the driver to be used in all subsequent control calls to the NI-488.2 driver. An example of the `OpenDriver` call is as follows:

```
osErr =  OpenDriver("\P.GPIB Driver", &refNum);
```

Note: ***A common Macintosh system error of*** `BdNumErr (-37)` ***returned by*** `OpenDriver` ***usually indicates that this routine has been given a bad string for the driver name. Refer to the documentation of the compiler being used to determine if a C string (NULL terminated) or a Pascal string (initial length byte) is needed for this Device Manager call.***

All other NI-488 functions are then accessed with control calls to the Device Manager. The `refNum` (returned in the `OpenDriver` call), control number, and handle to the parameter block are passed as parameters.

Making Control Calls

Each function has an identifying constant which you can use by including the header file `DrInterface.h`, which is found in the C LI folder. You can identify which function you are using by the constant name as shown in the following example:

```
osErr = Control(refNum, ibCAC, &paramBlk)
```

The constant name `ibCAC` indicates the function code sent to the driver.

Using Asynchronous Low-Level Device Manager Calls

You can use low-level asynchronous NI-488 and NI-488.2 Device Manager calls with optional completion routines if you are using the `NB Handler INIT` version 4.4 or later, or `NI-488 INIT` version 5.0 or later.

Set the asynchronous parameter of the low-level Device Manager call to `TRUE` to make the call asynchronous to the NI-488 driver. Refer to the Device Manager chapter of *Inside Macintosh*, Volume II, for more information on these parameters.

Note: ***Because of the limitations of the TCP interface, do not make any low-level asynchronous function calls which would directly or indirectly cause the GPIB-ENET bus to be put online or offline. These function calls include*** `ibbna`, `ibonl`, `ibfind`, ***and*** `ibdev`. ***Using these functions will result in synchronous behavior, but will not be destructive.***

Using Completion Routines

If you want to use a completion routine, place the address of the routine in the `ioCompletion` field of the low-level parameter block. Do *not* enter a null address; the function call would behave asynchronously and no completion routine would be executed.

You must save a copy of register A5 so that the completion routine can access it at execution time. This is accomplished by using `RememberA5()` to save a copy of the A5 register in the void function `GetA5()`. Place the `RememberA5()` code as the first statement in the main function body.

Use `SetUpA5()` in the beginning of the completion routine to retrieve the saved copy of the A5 register, and `RestoreA5()` at the end of the completion routine to restore A5 back to its original value.

Before the completion routine is called, register A0 is loaded with a pointer to the control block of the function performing the call. You can use the control block to determine which function call is completing.

The completion routine must preserve registers D2 through D7 and A2 through A4. Refer to the Device Manager chapter of *Inside Macintosh*, Volume II, for a more detailed explanation on which registers need to be preserved by completion routines.

The completion routine must not make any calls to the Memory Manager, either directly or indirectly, and cannot assume that any handles to unlocked blocks are valid. You need to evaluate your completion routine to determine whether it makes any calls that will directly or indirectly call the Memory Manager. National Instruments recommends against using any calls in the completion routine that are not specifically designated as safe by *Inside Macintosh*. Table B.1 in Appendix B of *Inside Macintosh*, Volume VI, lists the routines that may move or purge memory.

Printing Considerations

Considerations such as printing to the screen must be evaluated on a per-application basis. The application might be in the background and may not be the currently executing application. In such a case, printing directly to the application's window will cause adverse results.

Using Control Block Structures

Always use separate control block structures when you are making concurrent asynchronous calls to the NI-488.2 driver. If the completion routine has not been invoked from a previous asynchronous call, the main application code should not interfere with the fields inside the old control block, which invalidates the data in the control block structure for that call. The call may still be in the driver, and the field should not be altered until the completion routine for that function is called.

Invalid data also results from using a completion routine stack-based control block structure to make asynchronous calls from within the completion routine. An asynchronous call made in this manner fails because the completion routine may exit and deallocate the stack space *before* the call actually gets executed in the driver.

Calling ibrda, ibwrta, and ibcmda Asynchronously

The functions `ibrda`, `ibrwta`, and `ibcmda` have a unique definition regarding the execution of their completion routines if your application calls those functions asynchronously from the Device Manager.

In a typical application, `ibrda`, `ibwrta`, and `ibcmda` are called from the language interface and result in an immediate return of control to the user regardless of whether the I/O operation completed or not. Then `ibwait` is usually called to wait for the I/O to complete.

When these functions are called asynchronously from the Device Manager, any completion routine specified will execute immediately. This result occurs because, by definition, the *synchronous* portion of the call is complete even though the actual *asynchronous* portion of the call may not have completed.

If an `ibwait` is issued asynchronously through the Device Manager, its completion routine executes when the condition of the wait bits is satisfied.

National Instruments recommends against calling `ibrda`, `ibwrta`, and `ibcmda` asynchronously, even though they may function properly in theory. Instead, you can call `ibrd`, `ibwrt`, and `ibcmd` asynchronously from the Device Manager for most applications.

Parameter Block Structures and Examples

Parameters are passed to the device driver using a parameter block. The GPIB parameter block is `gpibBlock` (shown below using C syntax). `short` and `long` refer to 16-bit and 32-bit integers, respectively. A handle to this parameter block is passed as an argument to the high-level device driver Manager Control function. A pointer to this parameter block is the last field of the control parameter block (`MyCntrlParam`) passed to the low-level Device Manager `PBControl` function.

```
typedef struct StatusBlk{
      short  ibsta;
      short  iberr;
      short  ibret;          /* The four GPIB status variables   */
      long   ibcnt;
} StatusBlk;

typedef struct gpibBlock
{
      StatusBlk *statusBlk;
      short                  id;
      short   controlVar;    /* Control variable for some functions
      */
                             /* to indicate nature of action    */

      Ptr     IOBufPtr;      /* Pointer to the buffer in        */
                             /* user area, during I/O calls     */
      long   IOCount;        /* I/O byte count                  */
      short*  addr ;
      short*  result;
      short   limit ;
      void    (*srqservice) () ;
} gpibBlock;
```

The parameter block used for low-level device driver calls is a variant of the standard Control and Status parameter block. It is defined below as `MyCntrlParam`.

```
/*  parameter block for low-level Control/Status calls          */
typedef struct
{
      QElemPtr      qLink;
      Int16         qType;
      Int16         ioTrap;
      Ptr           ioCmdAddr;
      ProcPtr       ioCompletion;
      OsErr         ioResult;
      StringPtr     ioNamePtr;
      Int16         ioVRefNum;
      Int16         ioRefNum;/* driver routine ID              */
      Int16         csCode;
      Ptr           niParam; /* pointer to NBParamBlock
(gpibBlock)         */
      } MyCntrlParam;
typedef MyCntrlParam *MyCntrlPtr;
```

The last element, `niParam`, is a pointer to the function-specific parameter block, `gpibBlock`, as used in high-level Device Manager control calls.

Although the same parameter block, `gpibBlock`, is used for all NI-488 control functions, all parameters are not used by all functions. The *NI-488.2 Function Reference Manual for Macintosh* describes each function, the parameters used, their interpretation by the driver, and examples for calling each function (refer to the `LI.C` file). All examples are from the C language interface, using high-level Device Manager calls. All languages that permit toolbox calls can access the NI-488.2 driver in a similar manner.

Example Application (Low-Level Asynchronous with Completion Routines)

The following example program demonstrates how to correctly code your application to make asynchronous low-level Device Manager calls to the NI-488.2 driver. The sections following the example program explain segments of the program code in more detail.

The application attempts to read a chunk of data `TOTALBYTES` long in `MAXBUF` byte increments from a fictitious device at primary address 1. The completion routine continues to execute `ibrd`s until END is received in `ibsta`, or the total number of bytes read equals `TOTALBYTES`.

```
/*************************************************************/
/*                                                           */
/*               Example Program Using Asynchronous          */
/*                                                           */
/*                     Device Manager Calls                  */
/*                                                           */
/*************************************************************/

#include <Devices.h>         /* for OpenDriver()   */
#include <Events.h>          /* for GetNextEvent() */
#include <Stdio.h>           /* for printf()       */
```

```
#include "decl.h"            /* NI header file     */
#include "DrInterface.h"     /* NI header file     */

/*************************************************************/
/*                                                           */
/*                          Prototypes                       */
/*                                                           */
/*************************************************************/

static void GetA5             (void);
       void Completion_Routine(void);

/*************************************************************/
/*                                                           */
/*                          Constants                        */
/*                                                           */
/*************************************************************/

#define MAXBUF     50L         /* read block size     */
#define TOTALBYTES 1000000L    /* total bytes to read */

/*************************************************************/
/*                                                           */
/*  Function: static void GetA5(void):                       */
/*                                                           */
/*  Purpose:  Used in A5 register macros below.              */
/*                                                           */
/*************************************************************/

static void GetA5(void)
{
asm    {
           bsr.s     @LAB
           dc.l      0               ; Place A5 here
@LAB   move.l (sp)+,a1
       }
}

/*************************************************************/
/*                                                           */
/*  Macro:    RememberA5()                                   */
/*                                                           */
/*  Purpose:  Place the value of A5 into the GetA5() function*/
/*            body.                                          */
/*                                                           */
/*************************************************************/

#define RememberA5()           {                              \
                                    GetA5();                  \
                                    asm      {move.l A5,(a1)} \
                               }
```

```
/*************************************************************/
/*                                                           */
/*  Macro:    SetUpA5()                                      */
/*                                                           */
/*  Purpose:  Save the current value of A5 on the stack and  */
/*            retrieve the globals pointer A5 from the GetA5()  */
/*            function body.                                 */
/*                                                           */
/*************************************************************/

#define SetUpA5()               {                             \
                                     asm      {move.l A5,-(sp)}  \
                                     GetA5();                    \
                                     asm      {move.l (a1),A5 }  \
                                }

/*************************************************************/
/*                                                           */
/*  Macro:    RestoreA5()                                    */
/*                                                           */
/*  Purpose:  Retrieve the previous value of A5 from the stack  */
/*            and place it in register A5.                   */
/*                                                           */
/*************************************************************/

#define RestoreA5()             {                             \
                                     asm { move.l (sp)+,A5 }     \
                                }

/*************************************************************/
/*                                                           */
/*                    Application Globals                    */
/*                                                           */
/*************************************************************/

int                   brdID;    /* id of gpib0                      */
long              totalread;    /* total read                       */
char          buffer[MAXBUF];   /* read buffer                      */
Boolean     Finished = false;   /* loop control variable            */

/* Control Block for board setup calls */

MyNI488CntrlParam*     cr_cp;   /* pointer to control block         */
MyNI488CntrlParam cr_ctParam;   /* control block for comp rtn calls */
gpibBlock         cr_gpibBlk;   /* gpib parameter block             */
gpibBlock*     cr_gpibBlkPtr;   /* gpib parameter block pointer     */
StatusBlk          cr_status;   /* completion routine status block  */

/* Control Block for ibrd calls made from completion routine */

MyNI488CntrlParam*        cp;   /* pointer to control block         */
MyNI488CntrlParam    ctParam;   /* actual control block             */
gpibBlock            gpibBlk;   /* gpib parameter block             */
```

```
gpibBlock*        gpibBlkPtr;    /* gpib parameter block pointer      */
StatusBlk             status;    /* status block                      */

char*               drvrName =   (char*) "\p.GPIB Driver";

/*************************************************************/
/*                                                           */
/*  Function: Main                                           */
/*                                                           */
/*  Purpose:  Execute the following gpib commands using low- */
/*            level asynchronous device manager calls.       */
/*                                                           */
/*  Pseudo Code:                                             */
/*                                                           */
/*               brdID = ibfind("gpib0")                     */
/*                       ibsic(brdID)                        */
/*                       ibcmd(brdID,"A ",2L)                */
/*                       ibrd(brdID,buffer,MAXBUF)           */
/*                                                           */
/*               while(!done)                                */
/*               {                                           */
/*                 Give Processing Time to System            */
/*               }                                           */
/*               return to system                            */
/*                                                           */
/*************************************************************/

main ()
{
int                 osErr;
EventRecord        myEvent;

char*               cmd = "A ";             /* TA1 - LA0        */
char*               brdName = "gpib0";      /* bus name         */

RememberA5();

/* Set up gpib parameter block pointer */

gpibBlkPtr            = &gpibBlk;
cr_gpibBlkPtr         = &cr_gpibBlk;

/* Set up pointer to gpib status block */

gpibBlkPtr->statusBlk    = &status;
cr_gpibBlkPtr->statusBlk = &cr_status;

/* Set up pointer to gpib control block */

cp                    = &ctParam;
cr_cp                 = &cr_ctParam;

/* Setup pointer in the control block to the gpib parameter block */
```

```
cp->niParam           = (Ptr)gpibBlkPtr;
cr_cp->niParam        = (Ptr)cr_gpibBlkPtr;

/* Place a pointer to completion routine in the control block */

cp->ioCompletion      = (ProcPtr)Completion_Routine;
cr_cp->ioCompletion   = (ProcPtr)Completion_Routine;

/* Place a pointer to the driver name in the control block */

cp->ioNamePtr         = (StringPtr)drvrName;
cr_cp->ioNamePtr      = (StringPtr)drvrName;

/*************************************************************/
/*                                                           */
/* Open Console                                              */
/*                                                           */
/*************************************************************/

printf("Reading Data - Terminate on END or %ld bytes.\n\n",TOTALBYTES);

/*************************************************************/
/*                                                           */
/* Open the GPIB Driver to get the reference number          */
/*                                                           */
/*************************************************************/

osErr                 = PBOpen((ParmBlkPtr)cp,false);

if (osErr)
{
    printf("\nOpen error: %d\n", osErr);
    return;
}

/* copy driver reference number to the completion routine's control
block */

cr_cp->ioVRefNum = cp->ioVRefNum;
cr_cp->ioRefNum  = cp->ioRefNum;

/*************************/
/*                       */
/* brdID = ibfind("gpib0")  */
/*                       */
/*************************/

gpibBlkPtr->IOBufPtr= brdName;          /* ibFIND gpib0 */
cp->csCode                    = ibFIND;
osErr                         = PBControl((ParmBlkPtr)cp, true);
brdID                         = gpibBlkPtr->id;
```

```
/***************************/
/*                         */
/* ibsic(gpib0)            */
/*                         */
/***************************/

gpibBlkPtr->id              = brdID;
cp->csCode                  = ibSIC;
osErr                       = PBControl((ParmBlkPtr)cp, true);

/***************************/
/*                         */
/* ibcmd(gpib0,"A ",2L)    */
/*                         */
/***************************/

/* device at pad1 = tacs, gpib0 = lacs */

gpibBlkPtr->id              = brdID;
gpibBlkPtr->IOBufPtr        = cmd;
gpibBlkPtr->IOCount         = 2L;
cp->csCode                  = ibCMD;
osErr                       = PBControl((ParmBlkPtr)cp,true);

/*****************************************************************/
/*                                                               */
/* Initialize count variable                                     */
/*                                                               */
/*****************************************************************/

totalread = 0L;

/***************************/
/*                         */
/* ibrd(gpib0,buffer,MAXBUF)*/
/*                         */
/***************************/

gpibBlkPtr->id              = brdID;
gpibBlkPtr->IOBufPtr        = buffer;
gpibBlkPtr->IOCount         = MAXBUF;
cp->csCode                  = ibRD;
osErr                       = PBControl((ParmBlkPtr)cp,true);

/*****************************************************************/
/*                                                               */
/* Wait for TOTALBYTES bytes to come in. The completion routine */
/* will call ibrd asynchronously until END is detected.         */
/*                                                               */
/* Allow operating system to continue processing by calling OS  */
/* procedures GetNextEvent() and SystemTask().                   */
/*                                                               */
/*****************************************************************/
```

```
while(!Finished)
{
   SystemTask();
   GetNextEvent(everyEvent,&myEvent);
};

printf("\n");

printf("Total Count was %ld",totalread);

return;
}

/*************************************************************/
/*                                                           */
/*  Function: Completion_Routine                             */
/*                                                           */
/*  Purpose:  Execute ibrds of MAXBUF bytes until END is     */
/*            detected.                                      */
/*                                                           */
/*                                                           */
/*  Pseudo Code:                                             */
/*                                                           */
/*               while(!ibsta & END)                         */
/*               {                                           */
/*                       ibrd(brdID,buffer,MAXBUF);          */
/*               }                                           */
/*                                                           */
/*************************************************************/

void Completion_Routine()
{
int                  OsErr;      /* error codes */
int                myibsta;      /* local ibsta */
int                myiberr;      /* local iberr */
long               myibcnt;      /* local ibcnt */
MyNI488CntrlParam*      cp;      /* pointer to incoming control
block */
gpibBlock*      gpibBlkPtr;      /* gpib parameter block */

/*************************************************************/
/*                                                           */
/* Save registers d2-d7 and a2-a4                            */
/*                                                           */
/*************************************************************/

asm { movem.l d2-d7/a2-a4,-(a7) }

/*************************************************************/
/*                                                           */
/* Set register a5 to point to my applications global data area */
/*                                                           */
/*************************************************************/
```

```
SetUpA5();    /* Save the current A5 contents and then set register
                 A5 to its proper value for the calling application */
/*************************************************************/
/*                                                           */
/* Get pointer to control block for this call from register a0  */
/*                                                           */
/*************************************************************/

asm { move.l  a0,cp }

if(cp->csCode == ibRD)
{
       /* pick up pointer to local status block */

       gpibBlkPtr = (gpibBlockPtr)cp->niParam;

       myibsta    = gpibBlkPtr->statusBlk->ibsta;
       myiberr    = gpibBlkPtr->statusBlk->iberr;
       myibcnt    = gpibBlkPtr->statusBlk->ibcnt;

    totalread += myibcnt;

    /* If not END, read MAXBUF more bytes from the device */

       if(!(myibsta & END) && totalread<TOTALBYTES)
       {

        cr_cp->csCode             = ibRD;
        cr_gpibBlkPtr->id         = brdID;
        cr_gpibBlkPtr->IOBufPtr   = buffer;
        cr_gpibBlkPtr->IOCount    = MAXBUF;
        OsErr                     = PBControl((ParmBlkPtr)cr_cp, true);
       }
       else
       Finished = true;    /* notify main() that ibrds are finished */

}

/*************************************************************/
/*                                                           */
/* Restore A5 back to its old value                          */
/*                                                           */
/*************************************************************/

RestoreA5();   /* Restore A5 back to its original value */

/*************************************************************/
/*                                                           */
/* Restore registers d2-d7 and a2-a4                         */
/*                                                           */
/*************************************************************/
```

```
asm { movem.l (a7)+,d2-d7/a2-a4 }  /* Restore saved registers */

return;
}
```

Example Application Explanation

Main Body

The main body of the program performs the following steps in this order.

1. Calls `RememberA5()` to save a copy of the A5 register into the void function `GetA5()`.
2. Opens the GPIB driver and issues several function calls asynchronously to set up `gpib0` to be LACS and the device at pad 1 to be TACS.
3. Initializes control variables and total counts as it prepares to start reading data from the device.
4. Issues an `ibrd` of `MAXBUF` bytes on `gpib0`.
5. Waits, in its main loop, for the completion routine to set the state of the global variable `Finished` to TRUE. During this interim period, it continuously calls `GetNextEvent()` and `SystemTask()` to give the operating system sufficient processing time.
6. When `Finished` becomes TRUE, prints out the total number of bytes read and exits.

Completion Routine

The completion routine performs these steps in the following order.

1. Preserves registers D2 through D7 and A2 through A4.
2. Executes `SetUpA5()` to set the value of the A5 register to point to the global data area of the application.
3. Picks up a pointer to the original call's control block from register A0. The NI-488.2 driver places this value into register A0 to enable the completion routine to access status information on the recently completed function call.
4. Examines the control block of the calling function to determine whether the completion routine was called as a result of an `ibrd`. If so, a pointer to the gpib parameter block is extracted from the `cp->niParam` field of the control block. The routine evaluates the status information to determine whether the END condition has occurred. If END has *not* occurred and the total count is less than

`TOTALBYTES`, another `ibrd` call of `MAXBUF` bytes is issued to the device. Remember that the new `ibrd` call must use a *separate* control block structure. If END was detected, the global variable `Finished` is updated to inform the application's wait loop that the I/O is complete.

5. Restores the original contents of the A5 register with the `RestoreA5()` macro. Also restores registers D2 through D7 and A2 through A4 and returns control to the Device Manager.

For more Device Manager examples, refer to Chapter 3, *Device Manager Functions and Routines*, in the *NI-488.2 Function Reference Manual for Macintosh*.

Appendix D
Customer Communication

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

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

Corporate Headquarters
(512) 795-8248
Technical support fax: (800) 328-2203
(512) 794-5678

Branch Offices	**Phone Number**	**Fax Number**
Australia	(03) 879 9422	(03) 879 9179
Austria	(0662) 435986	(0662) 437010-19
Belgium	02/757.00.20	02/757.03.11
Denmark	45 76 26 00	45 76 71 11
Finland	(90) 527 2321	(90) 502 2930
France	(1) 48 14 24 00	(1) 48 14 24 14
Germany	089/741 31 30	089/714 60 35
Italy	02/48301892	02/48301915
Japan	(03) 3788-1921	(03) 3788-1923
Mexico	95 800 010 0793	95 800 010 0793
Netherlands	03480-33466	03480-30673
Norway	32-848400	32-848600
Singapore	2265886	2265887
Spain	(91) 640 0085	(91) 640 0533
Sweden	08-730 49 70	08-730 43 70
Switzerland	056/20 51 51	056/20 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	0635 523545	0635 523154

Technical Support Form

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

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

Name _____

Company _____

Address _____

_____

Fax (___) _____ Phone (___) _____

Computer brand _____

Model _____ Processor _____

Operating system _____

Speed _____MHz RAM _____MB

Display adapter _____

Mouse _____yes _____no

Other adapters installed _____

Hard disk capacity _____MB Brand _____

Instruments used _____

National Instruments hardware product _____

Revision _____

Configuration _____

National Instruments software product _____

Version _____

Configuration _____

(continues)

The problem is

List any error messages

The following steps will reproduce the problem

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: **NI-488.2™ User Manual for Macintosh**

Edition Date: **January 1995**

Part Number: **320897A-01**

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

(continues)

If you find errors in the manual, please record the page numbers and describe the errors.

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

Phone (_____) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway, MS 53-02
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
MS 53-02
(512) 794-5678

Glossary

Prefix	Meaning	Value
n-	nano-	10^{-9}
µ-	micro-	10^{-6}
m-	milli-	10^{-3}
k-	kilo-	10^{3}
M-	mega-	10^{6}

A

AC alternating current.

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

access board The GPIB board that controls and communicates with the devices on the bus that are attached to it.

ANSI American National Standards Institute.

ASCII American Standard Code for Information Interchange.

automatic serial polling (autopolling) A feature of the NI-488.2 software in which serial polls are executed automatically by the driver whenever a device asserts the GPIB SRQ line.

B

board-level function A rudimentary function that performs a single operation.

boot See *startup*.

C

CFE Configuration Enable is the GPIB command which precedes CFGn and is used to place devices into their configuration mode.

CFGn	These GPIB commands (CFG1 through CFG15) follow CFE and are used to configure all devices for the number of meters of cable in the system so that HS488 transfers occur without errors.
CIC	See *Controller-In-Charge*.
configuration	The process of altering the software parameters in the driver that describe characteristics of the devices and boards.
Controller-In-Charge (CIC)	The device that manages the GPIB by sending interface messages to other devices.
CPU	Central processing unit.

D

DAV (Data Valid)	One of the three GPIB handshake lines. See *handshake*.
DCL	Device Clear is the GPIB command used to reset the device or internal functions of all devices. See *SDC*.
dec	Decimal.
Device Clear	See DCL.
device-level function	A function that combines several rudimentary board operations into one function so that the user does not have to be concerned with bus management or other GPIB protocol matters.
DIO1 through DIO8	The GPIB lines that are used to transmit command or data bytes from one device to another.
DMA (direct memory access)	High-speed data transfer between the GPIB board and memory that is not handled directly by the CPU. Not available on some systems. See *programmed I/O*.
driver	Device driver software installed within the operating system.

E

END or END message	A message that signals the end of a data string. END is sent by asserting the GPIB End or Identify (EOI) line with the last data byte.

EOI (End or Identify)	A GPIB line that is used to signal either the last byte of a data message (END) or the parallel poll Identify (IDY) message.
EOS	End-of-string.
EOS byte	A 7- or 8-bit end-of-string character that is sent as the last byte of a data message.
EOT	End of transmission.
ESB	The Event Status bit is part of the IEEE 488.2-defined status byte which is received from a device responding to a serial poll.

F

FIFO	first-in-first-out.

G

GET	Group Execute Trigger is the GPIB command used to trigger a device or internal function of an addressed Listener.
Go To Local	See *GTL*.
GPIB	General Purpose Interface Bus is the common name for the communications interface system defined in ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1987.
GPIB address	The address of a device on the GPIB, composed of a primary address (MLA and MTA) and an optional secondary address (MSA). The GPIB board has both a GPIB address and an I/O address.
GPIB board	Refers to the National Instruments family of GPIB interface boards.
Group Executed Trigger	See *GET*.
GTL	Go To Local is the GPIB command used to place an addressed Listener in local (front panel) control mode.

H

handshake	The mechanism used to transfer bytes from the Source Handshake function of one device to the Acceptor Handshake function of another device. The three GPIB lines DAV, NRFD, and NDAC are used in an interlocked fashion to signal the phases of the transfer, so that bytes can be sent asynchronously (for example, without a clock) at the speed of the slowest device. For more information about handshaking, refer to the ANSI/IEEE Standard 488.1-1987.
hex	Hexadecimal; a number represented in base 16, for example decimal 16 = hex 10.
high-level function	See *device-level function.*
Hz	Hertz.

I

`ibcnt`	After each NI-488.2 I/O function, this global variable contains the actual number of bytes transmitted.
`iberr`	A global variable that contains the specific error code associated with a function call that failed.
`IBIC 488.2`	`IBIC 488.2`, the Interface Bus Interactive Control utility, is used to communicate with GPIB devices, troubleshoot problems, and develop your application.
`ibsta`	At the end of each function call, this global variable (status word) contains status information.
IEEE	Institute of Electrical and Electronic Engineers.
interface message	A broadcast message sent from the Controller to all devices and used to manage the GPIB.
I/O (Input/Output)	In the context of this manual, the transmission of commands or messages between the computer via the GPIB board and other devices on the GPIB.

I/O address	The address of the GPIB board from the point of view of the CPU, as opposed to the GPIB address of the GPIB board. Also called port address or board address.
`ist`	An Individual Status bit of the status byte used in the Parallel Poll Configure function.

K

KB	Kilobytes of memory.

L

LAD (Listen Address)	See *MLA*.
language interface	Code that enables an application program that uses NI-488 functions or NI-488.2 routines to access the driver.
listen address	See *MLA*.
Listener	A GPIB device that receives data messages from a Talker.
low-level function	See *board-level function*.

M

m	Meters.
MAV	The Message Available bit is part of the IEEE 488.2-defined status byte which is received from a device responding to a serial poll.
MB	Megabytes of memory.
memory-resident	Resident in RAM.
MLA (My Listen Address)	A GPIB command used to address a device to be a Listener. It can be any one of the 31 primary addresses.

MSA (My Secondary Address)	My Secondary Address is the GPIB command used to address a device to be a Listener or a Talker when extended (two byte) addressing is used. The complete address is a MLA or MTA address followed by an MSA address. There are 31 secondary addresses for a total of 961 distinct listen or talk addresses for devices.
MTA (My Talk Address)	A GPIB command used to address a device to be a Talker. It can be any one of the 31 primary addresses.

N

NDAC (Not Data Accepted)	One of the three GPIB handshake lines. See *handshake*.
`NI-488 Config`	The NI-488.2 driver configuration control panel utility.
NRFD (Not Ready For Data)	One of the three GPIB handshake lines. See *handshake*.

P

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

R

RAM	Random-access memory.
RQS	Request Service.

S

SC	See *System Controller*.
SDC	Selected Device Clear is the GPIB command used to reset internal or device functions of an addressed Listener. See *DCL* and *IFC*.
serial poll	The process of polling and reading the status byte of one device at a time. See *parallel poll*.
Service Request	See *SRQ*.
source handshake	The GPIB interface function that transmits data and commands. Talkers use this function to send data, and the Controller uses it to send commands. See *acceptor handshake* and *handshake*.
SPD (Serial Poll Disable)	Serial Poll Disable is the GPIB command used to cancel an SPE command.
SPE (Serial Poll Enable)	Serial Poll Enable is the GPIB command used to enable a specific device to be polled. That device must also be addressed to talk. See *SPD*.
SRQ (Service Request)	The GPIB line that a device asserts to notify the CIC that the device needs servicing.
startup	To load the operating system programs from floppy or hard disk into memory and to begin executing the code. A hard boot is when power is applied to the computer.
status byte	The IEEE 488.2-defined data byte sent by a device when it is serially polled.
status word	See *ibsta*.
System Controller	The single designated Controller that can assert control (become CIC of the GPIB) by sending the Interface Clear (IFC) message. Other devices can become CIC only by having control passed to them.

T

TAD (Talk Address)	See *MTA*.
Talker	A GPIB device that sends data messages to Listeners.
TCT	Take Control is the GPIB command used to pass control of the bus from the current Controller to an addressed Talker.
timeout	A feature of the NI-488.2 driver that prevents I/O functions from hanging indefinitely when there is a problem on the GPIB.
TLC	An integrated circuit that implements most of the GPIB Talker, Listener, and Controller functions in hardware.
TTL	Transistor-transistor logic.

U

ud (unit descriptor)	A variable name and first argument of each function call that contains the unit descriptor of the GPIB interface board or other GPIB device that is the object of the function.
UNL	Unlisten is the GPIB command used to unaddress any active Listeners.
UNT	Untalk is the GPIB command used to unaddress an active Talker.

Index

Numbers/Symbols

! Repeat Previous Function (IBIC 488.2), 4-15
$ Execute Indirect File function (IBIC 488.2), 4-16
+ Turn Display On function (IBIC 488.2), 4-15
- Turn Display Off function (IBIC 488.2), 4-15

A

address syntax, IBIC 488.2, 4-5
addressing, GPIB
    address bits (illustration), 1-2
    configuring in NI-488 Config utility
        Primary Address option, 6-8
        Repeat Addressing option, 6-11
        Secondary Address option, 6-8
        Unaddressing option, 6-10
    overview, 1-2
    repeat addressing, 3-3
AllSpoll routine, 5-7, 5-9
ANSI/IEEE Standard 488.1-1987, 1-1
application development. *See* debugging applications; programming.
Assert REN When System (Controller) option, NI-488 Config utility, 6-10
asynchronous low-level calls, Device Manager functions
    example application, C-5 to C-14
    how to use, C-2
    limitations, C-2
ATN (attention) line, 1-3
ATN status word condition, A-3
automatic serial polling. *See* serial polling.
auxiliary functions, IBIC 488.2
    ! (Repeat Previous Function), 4-15
    $ (Execute Indirect File), 4-16
    + (Turn Display On), 4-15
    - (Turn Display Off), 4-15
    Buffer (Set Buffer Display Mode), 4-17
    Help, 4-15
    list of functions (table), 4-14
    n* (Repeat Function n Times), 4-16
    PRINT (Display ASCII String), 4-16

B

BASIC. *See* QuickBASIC.
Buffer (Set Buffer Display Mode) function, IBIC 488.2, 4-17
bus/device frame, NI-488 Config utility
    bus only options, 6-9 to 6-11
    bus or device options, 6-8 to 6-9
    device only options, 6-11 to 6-12
    options (table), 6-7
Bus Timing option, NI-488 Config utility, 6-9 to 6-10
byte count, IBIC 488.2, 4-9

C

C language
    compiling, linking, and running applications, 2-17
    NI-488.2 software language files, 1-7
CIC protocol, for making GPIB board Controller-in-Charge, 5-3 to 5-4
CIC status word condition, A-3
CMPL status word condition, A-3
communication errors
    repeat addressing, 3-3
    termination method, 3-4
compiling, linking, and running applications
    C applications, 2-17
    QuickBASIC applications, 2-17 to 2-18
completion routines, Device Manager functions
    example application, C-13 to C-14
    how to use, C-2 to C-3
configuration. *See* GPIB operation; NI-488 Config utility.
configuration errors, debugging, 3-2
Configure Enable (CFE) message, 5-2
Configure (CFGn) message, 5-2
control block structures, Device Manager functions, C-3
control calls, Device Manager functions, C-2
control items, NI-488 Config utility, 6-4
Controllers
    CIC protocol for making GPIB board Controller-in-Charge, 5-3 to 5-4
    Controller-In-Charge and System Controller, 1-1
    device-level calls and bus management, 5-3 to 5-4
    GPIB operation, 1-1
    monitoring by Talker/Listener applications, 5-4
    System Controller option, NI-488 Config utility, 6-10
count, IBIC 488.2, 4-9
count variables (ibcnt and ibcntl), 2-5
customer communication, *xvi*, D-1

D

data lines, 1-2
data transfers
    high-speed (HS488), 5-2 to 5-3
        enabling, 5-2 to 5-3
        system configuration effects, 5-3
    terminating, 5-1 to 5-2
DAV (data valid) line, 1-3
DCAS status word condition
    description, A-4
    waiting for messages from Controller, 5-4
debugging applications. *See also* IBIC 488.2.
    common questions, 3-4 to 3-5
    communication errors
        repeat addressing, 3-3
        termination method, 3-4
    configuration errors, 3-2 to 3-3
    global status variables, 3-1
    GPIB error codes, 3-1 to 3-2
    IBIC 488.2, 3-1
    NI-488.2 Test, 3-1
    timing errors, 3-3
default configuration for NI-488.2 software, 6-3
device frame, NI-488 Config utility. *See* bus/device frame, NI-488 Config utility.
device-level calls and bus management, 5-3 to 5-4
Device Manager files, NI-488.2 software, 1-8
Device Manager interface
    asynchronous low-level Device Manager calls, C-2
    calling ibrda, ibwrta, and ibcmda asynchronously, C-3 to C-4
    completion routines, C-2 to C-3
    control block structures, C-3
    example application, C-5 to C-14
        completion routine explanation, C-13 to C-14
        main body explanation, C-13
    making control calls, C-2
    opening GPIB driver, C-1
    overview, C-1
    parameter block structures and examples, C-4 to C-5
    printing considerations, C-3
    when to use, 2-3, C-1
DMA option, NI-488 Config utility, 6-10
documentation
    conventions used, *xv*
    organization of, *xiv*
    related documentation, *xv-xvi*

driver, GPIB
    driver and driver utilities, NI-488.2, 1-6 to 1-7
    opening before using Device Manager functions, C-1
DTAS status word condition
    description, A-4
    waiting for messages from Controller, 5-4

E

EABO error code, B-4 to B-5
EADR error code, B-3
EARG error code, B-4
EBUS error code, B-6 to B-7
ECAP error code, B-6
ECIC error code, B-2
EDMA error code, B-5
EDVR error code, B-1 to B-2
EFSO error code, B-6
ELCK error code, B-8
END status word condition, A-2
ENEB error code, B-5
ENOL error code, B-3
EOI (end or identify) line
    definition (table), 1-3
    termination of data transfers, 5-1
EOIP error code, B-5 to B-6
EOS, configuring
    ibeos function, 5-1
    NI-488 Config utility
        EOS Byte option, 6-9
        EOS Modes option, 6-9
EOS comparison method, 5-1
EOS read method, 5-1
EOS write method, 5-1
ERR status word condition, A-2
error codes, IBIC 488.2 operation, 4-9
error codes and solutions
    debugging applications, 3-1 to 3-2
    EABO, B-4 to B-5
    EADR, B-3
    EARG, B-4
    EBUS, B-6 to B-7
    ECAP, B-6
    ECIC, B-2
    EDMA, B-5
    EDVR, B-1 to B-2
    EFSO, B-6

ELCK, B-8
ENEB, B-5
ENOL, B-3
EOIP, B-5 to B-6
ESAC, B-4
ESRQ, B-7 to B-8
ESTB, B-7
ETAB, B-8
list of error codes (table), 3-2, B-1
error variable (iberr), 2-5
errors, debugging
common questions, 3-4 to 3-5
communication errors
repeat addressing, 3-3
termination method, 3-4
configuration errors, 3-2
GPIB error codes, 3-1 to 3-2
timing errors, 3-3
ESAC error code, B-4
ESRQ error code, B-7 to B-8
ESTB error code, B-7
ETAB error code, B-8
Event Status bit (ESB), 5-5
Execute Indirect File function ($), IBIC 488.2, 4-16

F

FindRQS routine, 5-7, 5-8
functions. *See* Device Manager interface; IBIC 488.2; NI-488 functions.

G

General Purpose Interface Bus (GPIB). *See* GPIB operation.
global frame, NI-488 Config utility, 6-6 to 6-7
global variables
count variables (ibcnt and ibcntl), 2-5
debugging applications, 3-1
error variable (iberr), 2-5
status word (ibsta), 2-3 to 2-4, A-1 to A-4
GPIB addressing. *See* addressing, GPIB.
GPIB configuration utility. *See* NI-488 Config utility.
GPIB error codes. *See* error codes and solutions.

GPIB operation
- addressing, 1-2
- configuration
  - controlling more than one board, 1-5
  - linear and star configuration (illustration), 1-4
  - requirements, 1-5 to 1-6
- Controller-In-Charge and System Controller, 1-1
- interface management lines
  - ATN (attention), 1-3
  - EOI (end or identify), 1-3
  - IFC (interface clear), 1-3
  - REN (remote enable), 1-3
  - SRQ (service request), 1-3
- overview, 1-1
- sending messages, 1-2 to 1-3
- signals and lines
  - data lines, 1-2
  - DAV (data valid), 1-3
  - handshake lines, 1-3
  - NDAC (not data accepted), 1-3
  - NRFD (not ready for data), 1-3
- Talkers, Listeners, and Controllers, 1-1

GPIB programming techniques
- device-level calls and bus management, 5-3 to 5-4
- high-speed data transfers, 5-2 to 5-3
  - enabling HS488, 5-2 to 5-3
  - system configuration effects, 5-3
- parallel polling, 5-9 to 5-12
  - implementing, 5-9 to 5-12
  - using NI-488 functions, 5-10 to 5-11
  - using NI-488.2 routines, 5-11 to 5-12
- serial polling, 5-5 to 5-9
  - automatic serial polling, 5-5 to 5-7
    - autopolling and interrupts, 5-6
    - C "ON SRQ" capability, 5-6 to 5-7
    - stuck SRQ state, 5-6
  - service requests
    - from IEEE 488 devices, 5-5
    - from IEEE 488.2 devices, 5-5
  - SRQ and serial polling
    - with NI-488 device functions, 5-7
    - with NI-488.2 routines, 5-7 to 5-9
- Talker/Listener applications, 5-4
  - requesting service, 5-4
  - waiting for messages from Controller, 5-4
- termination of data transfers, 5-1 to 5-2
- waiting for GPIB conditions, 5-3

H

handshake lines, 1-3
help frame, NI-488 Config utility, 6-5
high-speed data transfers (HS488), 5-2 to 5-3
    enabling HS488, 5-2 to 5-3
    system configuration effects, 5-3
HS488. *See* high-speed data transfers (HS488).
HSS488 configuration message, 5-2

I

ibask function, 5-3
ibcmd function, 5-2
ibcmda function, calling asynchronously, C-3 to C-4
ibcnt and ibcntl count variables, 2-5
ibconfig function
    configuring GPIB board as CIC, 5-3
    determining assertion of EOI line, 5-1 to 5-2
    enabling autopolling, 5-5
    enabling high-speed data transfers, 5-2
ibdev function
    conducting parallel polls, 5-10
    IBIC 488.2, 4-10 to 4-11
ibeos function, 5-1
ibeot function, 5-1
iberr (error variable), 2-5
IBIC 488.2
    auxiliary functions
        ! (Repeat Previous Function), 4-15
        $ (Execute Indirect File), 4-16
        + (Turn Display On), 4-15
        - (Turn Display Off), 4-15
        Buffer (Set Buffer Display Mode), 4-17
        Help, 4-15
        list of functions (table), 4-14
        n* (Repeat Function n Times), 4-16
        PRINT (Display ASCII String), 4-16
        Set (Select Device or Board), 4-14
    byte count, 4-9
    debugging applications, 3-1
    error information, 4-9
    NI-488 functions commonly used with
        ibdev, 4-10 to 4-11
        ibfind, 4-10
        ibrd, 4-12
        ibwrt, 4-12

NI-488.2 routines commonly used with
examples, 4-1 to 4-4
Receive, 4-13
Send and SendList, 4-13
Set, 4-12
overview, 2-5, 4-1
status word (ibsta), 4-9
syntax
address syntax, 4-5
NI-488 functions, 4-5 to 4-7
NI-488.2 routines, 4-8
number syntax, 4-4
string syntax, 4-4 to 4-5
ibppc function
conducting parallel polls, 5-10
unconfiguring device for parallel polling, 5-11
ibrd function, 4-12
ibrda function, calling asynchronously, C-3 to C-4
ibrpp function, 5-11
ibrsp function, 5-6, 5-7
ibrsrv function, 5-4
ibsta. *See* status word (ibsta).
ibwait function
conducting serial polls, 5-7
Talker/Listener applications, 5-4
terminating stuck SRQ state, 5-6
waiting for GPIB conditions, 5-3
ibwrt function, 4-12
ibwrta function, calling asynchronously, C-3 to C-4
IFC (interface clear) line, 1-3
Interface Bus Interactive Control utility (IBIC 488.2). *See* IBIC 488.2.
interface management lines
ATN (attention), 1-3
EOI (end or identify), 1-3
IFC (interface clear), 1-3
REN (remote enable), 1-3
SRQ (service request), 1-3
interrupts and autopolling, 5-6

L

LACS status word condition
description, A-4
waiting for message from Controller, 5-4
lines. *See* signals and lines.

Listeners
    definition, 1-1
    Talker/Listener applications, 5-4
LOK status word condition, A-3

M

Message Available (MAV) bit, 5-5
messages, sending across GPIB, 1-2 to 1-3

N

n* Repeat Function n Times function (IBIC 488.2), 4-15
NDAC (not data accepted) line, 1-3
NI-488 applications, programming. *See* programming.
NI-488 Config utility
    Assert REN When System (Controller) option, 6-10
    bus/device frame, 6-7 to 6-11
        bus-only options, 6-9
        bus or device options, 6-8 to 6-9
        device-only options, 6-11
        options (table), 6-7
    Bus/Device menu, 6-2
    Bus Timing option, 6-9 to 6-10
    control items, 6-4
    default configuration, 6-3
    DMA option, 6-10
    EOS Byte option, 6-9
    EOS modes, 6-9
    exiting, 6-11 to 6-12
    global frame, 6-6 to 6-7
    help frame, 6-5
    Interface Type menu, 6-2
    opening, 6-1 to 6-2
    opening screen (illustration), 6-2
    overview, 6-1
    Primary Address pop-up menu, 6-8
    Rename Device option, 6-11
    Repeat Addressing option, 6-11
    Secondary Address pop-up menu, 6-8
    System Controller option, 6-10
    Timeout pop-up menu, 6-8
    TNT High Speed option, 6-10
    Unaddressing option, 6-10
    Use Bus option, 6-11

NI-488 functions
    board functions, 2-2
    device functions, 2-2
    one device per board concept, 2-1
    parallel polling, 5-10 to 5-11
    serial polling, 5-7
    using in IBIC 488.2
        ibdev, 4-10 to 4-11
        ibfind, 4-10
        ibrd, 4-12
        ibwrt, 4-12
        Set, 4-12
        syntax (table), 4-6 to 4-7
NI-488.2 applications, programming. *See* programming.
NI-488.2 routines
    capabilities, 2-2
    parallel polling, 5-11 to 5-12
    serial polling, 5-7 to 5-9
    serial polling examples
        AllSpoll, 5-9
        FindRQS, 5-8
    using in IBIC 488.2
        Receive, 4-13
        Send, 4-13
        SendList, 4-13
        Set, 4-12
        syntax (table), 4-8
NI-488.2 software
    C language files, 1-7
    default configuration, 6-3
    Device Manager files, 1-8
    driver and driver utilities, 1-6 to 1-7
    how NI-488.2 works with your system, 1-8
    QuickBASIC language files, 1-7
NI-488.2 Test utility, 3-1
NRFD (not ready for data) line, 1-3
number syntax, IBIC 488.2, 4-4

O

operation of GPIB. *See* GPIB operation.

P

parallel polling, 5-9 to 5-12
    implementing, 5-9 to 5-12
    using NI-488 functions, 5-10 to 5-11
    using NI-488.2 routines, 5-11 to 5-12
parameter block structures, Device Manager functions, C-4 to C-5
PPoll routine, 5-11
PPollConfig routine, 5-11
PPollUnconfig routine, 5-12
primary GPIB address
    definition, 1-2
    setting in NI-488 Config utility, 6-8
PRINT (Display ASCII String function, IBIC), 4-16
printing considerations, Device Manager functions, C-3
programming. *See also* debugging applications; GPIB programming techniques.
    checking status with global variables
        count variables (ibcnt and ibcntl), 2-5
        error variable (iberr), 2-5
        status word (ibsta), 2-3 to 2-4
    choosing programming method
        Device Manager, 2-3
        NI-488.2 language interface, 2-1 to 2-2
    compiling, linking, and running
        C applications, 2-17
        QuickBASIC applications, 2-17 to 2-18
    examples
        Device Manager interface, C-5 to C-14
            completion routine explanation, C-13 to C-14
            main body explanation, C-13
        NI-488.2 routines in IBIC 488.2, 4-1 to 4-4
    IBIC 488.2 for communicating with devices, 2-5
    NI-488 applications
        clearing devices, 2-8
        configuring devices, 2-8 to 2-9
        items to include, 2-6
        NI-488 program shell (illustration), 2-7
        opening devices, 2-8
        placing device offline, 2-10
        processing data, 2-10
        reading measurements, 2-10
        triggering devices, 2-9
        waiting for measurements, 2-9 to 2-10
    NI-488.2 applications
        configuring instruments, 2-15
        finding all Listeners, 2-13
        identifying instruments, 2-13 to 2-14
        initialization, 2-13

initializing instruments, 2-14
items to include, 2-11
NI-488.2 program shell (illustration), 2-12
placing board offline, 2-16
processing data, 2-16
reading measurements, 2-16
triggering instruments, 2-15
waiting for measurements, 2-15 to 2-16

Q

QuickBASIC
compiling, linking, and running applications, 2-17 to 2-18
NI-488.2 software language files, 1-7

R

ReadStatusByte routine, 5-7
Receive routine, IBIC 488.2, 4-13
REM status word condition, A-3
REN (remote enable) line, 1-3
Rename Device option, NI-488 Config utility, 6-11
repeat addressing
enabling in NI-488 Config utility, 6-11
required before GPIB activity, 3-3
Repeat Function n Times (n*), IBIC 488.2, 4-16
Repeat Previous Function (!), IBIC 488.2, 4-15
RQS status word condition, A-2 to A-3

S

secondary GPIB address
definition, 1-2
setting in NI-488 Config utility, 6-8
Send routine, 4-13
SendCmds function, 5-2
sending messages across GPIB, 1-2 to 1-3
SendList routine, 4-13
serial polling, 5-5 to 5-9
automatic serial polling, 5-5 to 5-7
autopolling and interrupts, 5-6
C "ON SRQ" capability, 5-6 to 5-7
stuck SRQ state, 5-6

service requests
from IEEE 488 devices, 5-5
from IEEE 488.2 devices, 5-5
Talker/Listener applications, 5-4
SRQ and serial polling
with NI-488 device functions, 5-7
with NI-488.2 routines, 5-7 to 5-9
service requests
serial polling
IEEE 488 devices, 5-5
IEEE 488.2 devices, 5-5
stuck SRQ state, 5-6
Talker/Listener applications, 5-4
Set function (Select Device or Board), IBIC 488.2, 4-14
Set routine, IBIC 488.2, 4-12
signals and lines
ATN (attention), 1-3
data lines, 1-2
DAV (data valid), 1-3
EOI (end or identify), 1-3
handshake lines (table), 1-3
IFC (interface clear), 1-3
interface management lines (table), 1-3
NDAC (not data accepted), 1-3
NRFD (not ready for data), 1-3
REN (remote enable), 1-3
SRQ (service request), 1-3
SRQ (service request) line
definition, 1-3
serial polling
automatic serial polling, 5-5 to 5-6
C "ON SRQ" capability, 5-6 to 5-7
stuck SRQ state, 5-6
using NI-488 device functions, 5-7
using NI-488.2 routines, 5-7 to 5-9
SRQI status word condition, A-2
status word (ibsta)
ATN, A-3
CIC, A-3
CMPL, A-3
DCAS, A-4
DTAS, A-4
END, A-2
ERR, A-2
IBIC 488.2 operation, 4-9
LACS, 5-4, A-4
list of status word bits (table), 2-4, A-1
LOK, A-3
REM, A-3

RQS, A-2 to A-3
SRQI, A-2
TACS, 5-4, A-4
testing for ibsta conditions, 2-3 to 2-4
TIMO, A-2
string syntax, IBIC 488.2, 4-4 to 4-5
stuck SRQ state, 5-6
syntax, IBIC 488.2. *See* IBIC 488.2.
System Controller
configuring in NI-488 Config utility, 6-10
GPIB operation, 1-1

T

TACS status word condition
definition, A-4
waiting for message from Controller, 5-4
Talker/Listener applications
definition, 5-4
requesting service, 5-4
waiting for messages from Controller, 5-4
Talkers, 1-1
technical support, D-1
termination of data transfers
debugging applications, 3-4
GPIB programming techniques, 5-1 to 5-2
TestSRQ routine, 5-7
timeout value, setting in NI-488 Config utility, 6-8
timing errors, 3-3
TIMO status word condition, A-2
TNT High Speed option, NI-488 Config utility, 6-10
TNT4882C hardware, 5-2
Turn Display Off function (-), IBIC 488.2, 4-15
Turn Display On function (+), IBIC 488.2, 4-15

U

Unaddressing option, NI-488 Config utility, 6-10
Use Bus option, NI-488 Config utility, 6-11

W

WaitSRQ routine, 5-7
writing applications. *See* programming.