GPIB

NI-488DDK™ Software Reference Manual

NATIONAL INSTRUMENTS™

January 2003 Edition
Part Number 321418B-01

Worldwide Technical Support and Product Information

`ni.com`

National Instruments Corporate Headquarters

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

Worldwide Offices

Australia 61 2 9672 8846, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599, Canada (Calgary) 403 274 9391, Canada (Montreal) 514 288 5722, Canada (Ottawa) 613 233 5949, Canada (Québec) 514 694 8521, Canada (Toronto) 905 785 0085, Canada (Vancouver) 514 685 7530, China 86 21 6555 7838, Czech Republic 42 02 2423 5774, Denmark 45 45 76 26 00, Finland 385 0 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, Greece 30 2 10 42 96 427, Hong Kong 2645 3186, India 91 80 4190000, Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400, Malaysia 603 9059 6711, Mexico 001 800 010 0793, Netherlands 31 0 348 433 466, New Zealand 64 09 914 0488, Norway 47 0 32 27 73 00, Poland 48 0 22 3390 150, Portugal 351 210 311 210, Russia 7 095 238 7139, Singapore 65 6 226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 2 2528 7227, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on the documentation, send email to `techpubs@ni.com`.

Important Information

Warranty

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

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

National Instruments believes that the information in this document 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

HS488™, National Instruments™, NI™, NI-488™, NI-488.2™, NI-488DDK™, and ni.com™are trademarks of National Instruments Corporation.

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

Patents

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

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Contents

About This Manual

Chapter 1
Introduction

Chapter 2
Developing Your Driver

Chapter 3
Developing Your Application

Chapter 4
NI-488DDK Functions

Chapter 5
GPIB Programming Techniques

Appendix A
Multiline Interface Messages

Appendix B
Status Word Conditions

Appendix C
Error Codes and Solutions

Appendix D
Technical Support and Professional Services

Glossary

Index

About This Manual

This manual describes the features and functions of the NI-488 Driver Development Kit (NI-488DDK) software. You can customize the NI-488DDK software for the operating system you use. This manual assumes that you are already familiar with general operating system fundamentals and device driver development concepts.

Conventions

The following conventions appear in this manual:

This icon denotes a note, which alerts you to important information.

italic — Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.

`monospace` — Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.

`monospace bold` — Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.

`monospace italic` — Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

IEEE 488 and IEEE 488.2 — *IEEE 488* and *IEEE 488.2* refer to the ANSI/IEEE Standard 488.1-1987 and the ANSI/IEEE Standard 488.2-1992, respectively, which define the GPIB.

paths — Paths in this manual are denoted using backslashes (\) or forward slashes (/) to separate drive names, directories, folders, and files.

Related Documentation

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

- ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation*
- ANSI/IEEE Standard 488.2-1992, *IEEE Standard Codes, Formats, Protocols, and Common Commands*

1

Introduction

This chapter describes the NI-488DDK software and gives an overview of GPIB.

NI-488DDK Software

The NI-488DDK software provides a subset of the GPIB functionality found in standard NI-488.2 drivers from National Instruments. It is intended primarily for use by customers who need to develop GPIB applications on computers or operating systems for which standard NI-488.2 kits are not available.

The application programming interface (API) of the NI-488DDK software is completely compatible with the API of standard NI-488.2 drivers. However, internally, the NI-488DDK software is designed to be easily customized by users familiar with device driver development. The design is compatible with a variety of modern operating systems, including both singlethreaded and multithreaded kernels. For your convenience, the software distribution includes one or more example OS-specific implementations that you can use with little or no change on the intended operating system. You can also use these implementations as templates for developing your own OS-specific implementations.

Working with the Distribution Media

The NI-488DDK software is distributed on a CD-ROM. When distributed electronically, the software is provided in a single, compressed, `tar`-formatted file compatible with most UNIX-based systems. The top-level contents of the CD-ROM are as follows:

`_README_`	Version specific documentation file
`DDK_TAR.Z`	Alternate distribution file for UNIX users
`DRIVER`	Driver source files directory
`UTIL`	Utility source files directory
`PATENTS.TXT`	National Instruments patents information file

The DDK_TAR.Z file contains the complete NI-488DDK software distribution in a compressed, tar format that UNIX users and others may find easier to work with when transferring the software to other systems. To extract the distribution files from DDK_TAR.Z, or from a similar file received electronically, transfer the file without conversion (for example, if using FTP, transfer the file in binary mode) to a UNIX-compatible system and enter the following commands:

```
uncompress DDK_TAR.Z
tar xvf DDK_TAR
```

After the contents of the file have been extracted, the current directory contains a distribution directory named ni488ddk_v*X.X*, where *X.X* is a version number. The contents of the extracted distribution directory are as follows:

README	Version-specific documentation file
driver/	Driver source files directory
util/	Utility source files directory

Working with the Distribution Contents

The _README_ file contains any additional information or changes to the software documentation made since this manual was last updated.

The driver/ directory on the distribution media contains all of the source files necessary to build the NI-488DDK driver. To port the DDK driver to a specific operating system, you generally need to modify only the files in one of the driver/OS_Layer/ subdirectories. This process is described in greater detail in Chapter 2, *Developing Your Driver*.

The util/ directory on the distribution media contains the C source files for a sample IEEE 488.2 application library, as well as an automated driver test program and other diagnostic tools. You can use the 488.2 application library to provide a layer of high-level GPIB functionality to your application and to see how to use some of the low-level functions in the DDK driver. You can use the driver test program to verify that your DDK driver is working properly after you have written the files in the new OS Layer directory. Use of the driver test program and other diagnostic tools is described in greater detail in Chapter 2, *Developing Your Driver*. Use of the IEEE 488.2 application library is described in Chapter 3, *Developing Your Application*. Some of the IEEE 488.2 routines provided in the application library for SRQ servicing are described in Chapter 5, *GPIB Programming Techniques*. For information about other routines provided in the IEEE 488.2 library, refer to the ni4882.c file in the util/ directory.

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 Mbyte/s and above, using a 3-wire handshake. The bus supports one System Controller, usually a computer, and up to 14 additional instruments. The ANSI/IEEE Standard 488.2-1992 extends IEEE 488.1 by defining a bus communication protocol, a common set of data codes and formats, and a generic set of common device commands.

Talkers, Listeners, and Controllers

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

Some devices are capable of playing more than one role. A digital voltmeter, for example, can be a Talker and a Listener. If your personal computer has a National Instruments GPIB interface board and NI-488DDK software installed, it can function as a Talker, Listener, and Controller.

Controller-In-Charge and System Controller

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

GPIB Addressing

All GPIB devices and boards must be assigned a unique GPIB address. A GPIB address is made up of two parts: a primary address and an optional secondary address.

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

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

Table 1-1. GPIB Address Bits

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

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 1-2 summarizes the GPIB handshake lines.

Table 1-2. GPIB Handshake Lines

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

Interface Management Lines

Five GPIB hardware lines manage the flow of information across the bus. Table 1-3 summarizes the GPIB interface management lines.

Table 1-3. GPIB Interface Management Lines

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

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 1-1 shows the linear and star configurations.

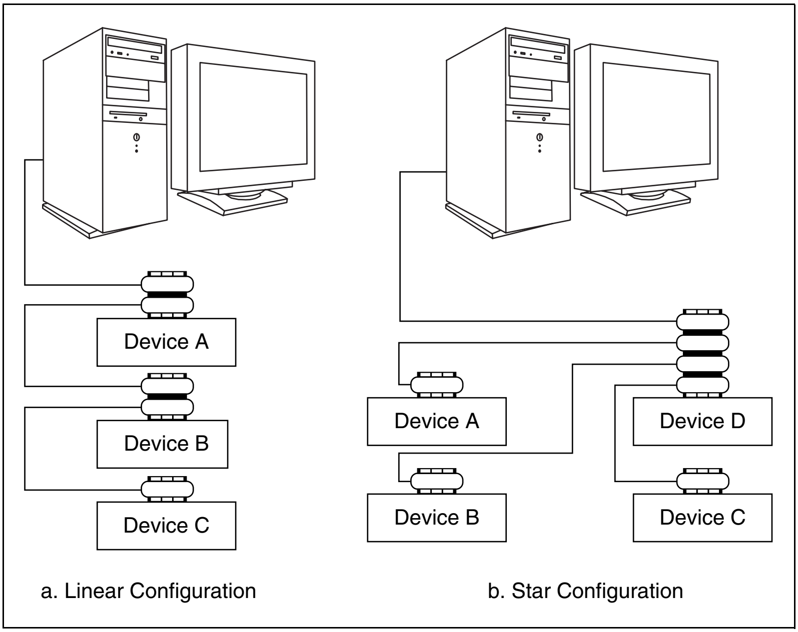

Figure 1-1. Linear and Star System Configuration

Controlling More Than One Board

Figure 1-2 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.

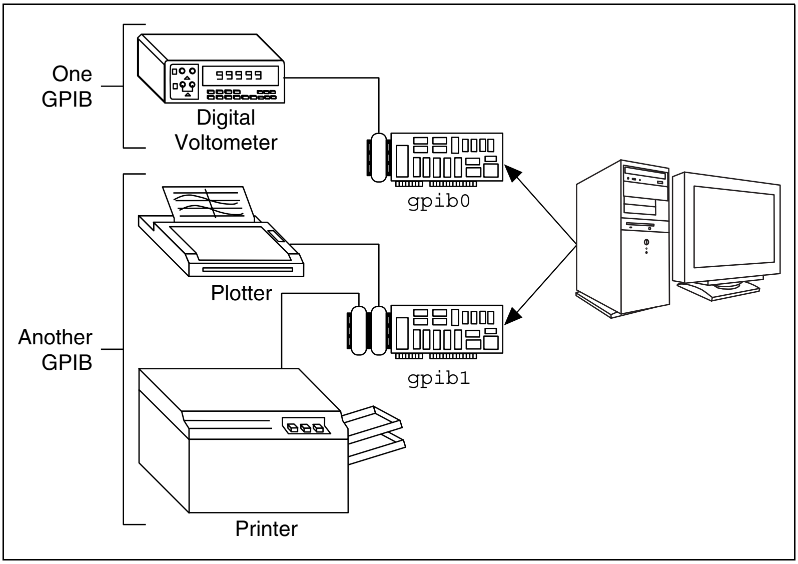

Figure 1-2. Example of Multiboard System Setup

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

For high-speed operation, the following restrictions apply:

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

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

2

Developing Your Driver

This chapter describes the organization of the NI-488DDK driver and provides guidelines for developing and debugging the driver on a particular operating system.

Driver Organization

The NI-488DDK driver consists of a low-level driver module (IB) and a high-level language interface module (CIB). The IB module includes three separate layers: an Operating System Layer, Common Layer, and Hardware Layer. The CIB module includes only two layers: an Operating System Layer and Common Layer. The organization of the files in the `driver/` directory on the distribution media, as shown in Table 2-1, reflects the organization of both the IB and CIB modules.

For a list of the specific files included in a particular DDK release, refer to the `_README_` file in the main `driver/` directory and to the `_README_` files, if present, in the `HW_Layer/` and `OS_Layer/` subdirectories.

The IB module (`ib.c`) and CIB module (`cib.c`) are organized as shown in Figures 2-1 and 2-2, respectively. As indicated in Figure 2-1, the `ib.c` file serves as a container, among other things, for all the other files that make up the low-level driver. Likewise, the `cib.c` file serves as a container for the files that make up the C language interface. The `ib.c` and `cib.c` files are described in the *Writing a New OS Layer* section of this chapter.

Table 2-1. NI-488DDK Driver Directory

Functional Layer	File	Description
Hardware Dependent (HW_Layer/)	_README_	Hardware-specific documentation file (optional)
	nichp_hw.h	General chip-level include file
	ni<Bus Type>_hw.c	Bus-specific source files
	ni<Bus Type>_hw.h	Bus-specific include files
	nitn_chw.c	TNT chip-specific source file
	nitn_chw.h	TNT chip-specific include file
Operating System Dependent (OS_Layer/*/)	_README_	OS-specific documentation file (optional)
	cib.c	OS-specific C Language/API source file
	cib.h	OS-specific C Language/API include file
	ib.c	OS-specific driver source file
	ib.h	OS-specific driver include file
	makefile	OS/compiler-specific driver make file
Common	_README_	Version-specific documentation file
	cibgen.c	Generic C language interface source file
	ibconf.h	Driver configuration include file
	ni488.c	NI-488DDK functions source file
	ni_proto.h	Prototype include file
	ni_suprt.c	Support functions source file
	ni_suprt.h	Support functions include file
	ugpib.h	User application include file

```
/******************************************************
 * NI-488 Driver Development Kit for GPIB Interfaces
 * Copyright (c) 1997-2003 National Instruments Corporation
 * All rights reserved.
 ******************************************************/
 :
 :
#include "cib.h"                /* NI include files...  */
#include "ugpib.h"
#include "ibconf.h"
#include "ib.h"
#include "ni_suprt.h"
#include "nichp_hw.h"
#include "nivme_hw.h"
#include "ni_proto.h"
 :
 :
#include "ni_suprt.c"           /* Generic IB code...   */
#include "ni488.c"

#include "nivme_hw.c"           /* HW-specific code     */
 :
 :
```

Figure 2-1. The IB Driver Module (ib.c)

```
/******************************************************
 * NI-488.2 C Language Interface
 * Copyright (c) 1997-2003 National Instruments Corporation
 * All rights reserved.
 ******************************************************/
 :
 :
#include "cib.h"
#include "ugpib.h"
 :
 :
#include "cibgen.c"             /* Generic CIB code     */
 :
 :
```

Figure 2-2. The CIB Language Interface Module (cib.c)

Driver Coding Conventions

Following are some of the coding conventions adopted throughout the source files of the NI-488DDK driver. You may find it useful to keep the following conventions in mind when studying the driver source code, and when adding new code of your own:

- The names of all C functions and variables begin with a lowercase letter, and may contain both uppercase and lowercase letters.
- The names of all C macros begin with an uppercase letter, and may contain both uppercase and lowercase letters.
- The names of all `#define` constants are composed of only uppercase letters.
- With the exception of the `cibgen.c` source file, all functions within the same C source file begin with the prefix of the file name itself. For example, all functions in the `ib.c` file begin with the prefix `ib_`, and all functions in the `ni_suprt.c` file begin with the prefix `ni_`.

Choosing an Implementation Method

Depending on your target operating system, you can implement the NI-488DDK driver as a user or kernel-level driver. You can link a user-level driver directly to your application, just as you would any other object file or library. You install a kernel-level driver as part of the operating system, thus making it a system resource available to all application programs. In general, user-level drivers are easier to implement than kernel-level drivers. Some operating systems support either method, while others support only the kernel-level method.

The implementation method you choose may depend on several factors. For example, a user-level implementation may be adequate if the driver is used by only one application at a time and the driver does not use interrupts. Conversely, a kernel-level implementation may be necessary if the driver must be shared among several applications or if you want interrupt support. There may also be performance issues related to either implementation choice. Refer to the driver development documentation for your target operating system for more information about your options.

In a user-level implementation, all of the `.c` files in the target `OS_Layer` subdirectory are linked to the application program, either directly or indirectly. In a kernel-level implementation, only the `cib.c` file is linked to the application program, while the `ib.c` file is linked into the operating system kernel. Refer to Figures 2-3 and 2-4 for illustrations of the differences between these two implementation methods.

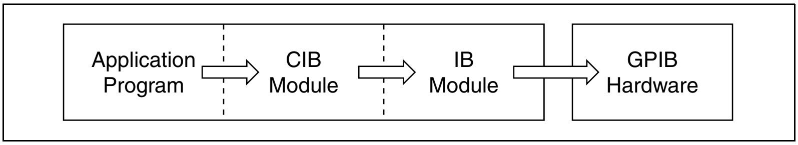

Figure 2-3. User-Level Implementation

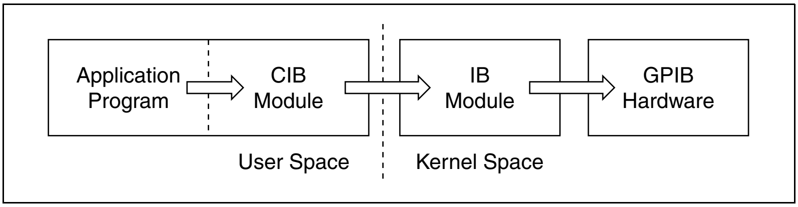

Figure 2-4. Kernel-Level Implementation

Writing a New OS Layer

Regardless of the implementation method you choose, porting the NI-488DDK driver primarily involves writing a new OS Layer for the target operating system. The OS Layer contains all of the system-specific code necessary to interface the application program to the DDK driver, and to interface the driver to the system.

Support Code Location

The code that interfaces the application program to the DDK driver is contained in the `cib.*` files. These files make up the OS-dependent portion of the C language interface to the driver. The generic, OS-independent portion of the C language interface is in the `cibgen.c` file of the main `driver/` directory.

The low-level OS interface to the other layers of the DDK driver is provided through a set of macros and data types defined in the `ib.h` file, with additional support code provided in the `ib.c` file as needed. This support code includes several constants, variables, and functions that are required in all NI-488DDK implementations. The `ib.c` file also provides any other OS-specific functionality required for the operation of the driver, such as initializing the driver, registering interrupts, and calling the ni488_enter entry point after you call the driver via its standard entry point. On UNIX and some other operating systems, this entry point is typically ioctl.

A listing and brief description of the constants, macros, functions, and data types required in the OS Layer files of the driver are provided in the _README_ file of the main driver/ directory. For a better understanding of the usage and purpose of the items listed there, refer to the code and comments of the C files in the example driver/OS_Layer/ subdirectories.

Porting the DDK Driver

The easiest way to port the DDK driver to a new OS is to copy the files in one of the example OS Layer subdirectories to a new subdirectory and then modify those files for the new OS. Choose an OS Layer implementation that is most like the one you want to develop for the target operating system. In some cases, it may be necessary to divide the functionality contained in one of the .c files into two or more files. For example, if some of the ib.c functionality must be written in assembly language, you might decide to set up two ib files, ib.c and ib.asm. If at all possible, avoid modifying any driver files outside of the OS Layer unless there is a compelling reason to do so (for example, to fix a bug). By limiting your changes to the OS files only, you ensure maximum source code compatibility with any future versions of the DDK package, as well as functional compatibility with other NI-488 drivers from National Instruments.

Note National Instruments generally cannot provide support for or answer questions about a specific OS to which you are trying to port. For specific questions regarding the features and functionality of a target OS, contact the OS vendor.

Compiling, Linking, and Installing the Driver

After editing the files in the driver/OS_Layer subdirectory, you create an executable NI-488DDK driver by compiling the ib.c file. To create the C language interface to the driver, compile the cib.c file. Refer to the documentation that came with the operating system and the C compiler you are using for detailed information about compiling, linking, and installing a new device driver. You might also find it useful to refer to the makefile files included in the example OS_Layer subdirectories on the distribution media.

In general, you compile the `ib.c` and `cib.c` source files the same way you would any other C source file, to produce two binary object files. If you are implementing a user-level driver, in most cases you should link both the `ib` and `cib` object files directly to your application program. If you are implementing a kernel-level driver, only the `cib` object file should be linked directly to your application, while the `ib` object file must be linked into the operating system kernel itself. Depending on your system, you link the `ib` object file into the system by either statically linking it to the object files that make up the system kernel, and then rebooting the system, or by executing special system commands to load the `ib` object file dynamically into the system, without rebooting.

Testing and Debugging the Driver

The NI-488DDK distribution media includes an automated test program called `ibchat` that you can use to verify the correct operation of a new NI-488DDK driver. The program is written in C and is compatible with a variety of text-based systems. The test is designed to be run between two GPIB interfaces installed in separate NI-488 based systems, or between two GPIB interfaces installed in a single, multitasking, NI-488 based system.

Separate invocations of `ibchat` must be run on both GPIB interfaces participating in the test, but both interfaces do not have to be controlled by an NI-488DDK driver. Because `ibchat` is an NI-488 application, it can be run using any NI-488 compatible driver, which gives you flexibility in setting up the test. For example, for one side of the test, you could compile and run `ibchat` on any of a variety of Windows or UNIX-based systems supported by NI-488.2 drivers from National Instruments.

To use the test program, compile and link `ibchat.c` as you would any other NI-488DDK application, and run the resulting executable file. This process is described in Chapter 3, *Developing Your Application*. For example, on a UNIX-based system, you might enter the following commands to compile and run `ibchat`:

```
cc ibchat.c cib.c -o ibchat
ibchat
```

After startup, `ibchat` prompts you to designate one interface at address 0 (`MA0`) and the other interface at address 1 (`MA1`). The test supports a number of command line options that you can use to modify the behavior of the test. These options are described briefly in an online help screen that you can access by starting the program as follows:

```
ibchat -h
```

Depending on the options selected, the `ibchat` test runs until it completes or until you terminate it manually. If the test encounters an error before terminating normally, the test halts and prints out some diagnostic information to help you determine the nature of the error.

Debugging Run-Time Errors

In addition to the `ibchat` diagnostic information, you can make use of an extensive set of conditional debugging and tracing statements available in the DDK driver source files to help identify and fix run-time errors. These statements can be configured via compile-time and run-time flags to direct debugging information to a system console, an internal tracing buffer in the driver, or to an extra GPIB interface connected to a GPIB analyzer.

Console print statements are the easiest debugging statements to use, but they are also the slowest. These statements are best suited for low-speed testing, such as when you intend to step through the driver testing one function at a time.

Debugging statements written to an internal trace buffer within the driver are much faster than console print statements, and are therefore better suited for higher speed testing of longer duration. You can retrieve the contents of the internal trace buffer at any time by using the `ibdump` extraction program, included in the NI-488DDK distribution.

You can also arrange to have the contents of the internal trace buffer output to an unused GPIB interface as they are generated at run time. This is another flexible option that is suitable for high-speed testing situations. This option is especially useful when debugging a problem that is causing the system to crash (a common problem in driver development) before the contents of the internal trace buffer can be retrieved using `ibdump`.

To conditionally include the debugging statements, first set the `GPIB_DEBUG` and (optionally) the `GPIB_TRACE` flags in the `ib.c` file to `1`. Once the `DBG` statements have been compiled into the driver, you can use the `ibpoke` driver function to control the quantity and display options of these statements at run time. The `ibdump` program is written in C and can be compiled as either a separate utility or as a linkable subroutine (suitable for use with user-level drivers) that you can call from your application program.

Documentation of Debugging Tools

For more information about the debugging options available in a particular version of the NI-488DDK driver, refer to the definitions at the top of the `ni_suprt.h` file. For information about the `ibpoke` function, refer to Chapter 4, *NI-488DDK Functions*. For more information about the `ibchat` and `ibdump` programs, refer to the comments in their respective C source files. For information about other tools and techniques that may be available to help debug your NI-488DDK driver, depending on your development environment, refer to the documentation that came with your operating system.

3

Developing Your Application

This chapter explains how to develop a GPIB application using NI-488DDK functions.

Using NI-488DDK Functions

NI-488DDK functions perform only rudimentary GPIB operations. These low-level functions access the interface board directly and require you to handle the addressing and bus management protocol. In addition to serving as a foundation upon which you can implement higher-level functions, these functions give you the flexibility and control to handle situations such as the following:

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

The NI-488DDK functions are compatible with the corresponding functions of standard NI-488.2 drivers from National Instruments. However, the NI-488DDK may not have all the functionality of a standard NI-488.2 driver. Refer to Chapter 4, *NI-488DDK Functions*, for details.

Items to Include in Your Application

Items you should include in your C application programs are as follows:

- The header file `ugpib.h` contains prototypes for the GPIB functions and constants that you can use in your application.
- One or more calls to the `ibfind` function to obtain a unit descriptor for each GPIB board that the application uses.
- Code to check for errors after each NI-488DDK function call.
- A function to handle GPIB errors. This function takes the board offline and closes the application. If the function is declared as:

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

then your application invokes it as follows:

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

Checking Status with Global Variables

Each NI-488DDK function updates four global variables to reflect the status of the board that you are using. These global status variables are the status word (`ibsta`), the error variable (`iberr`) and the count variables (`ibcnt` and `ibcntl`). They contain useful information about the performance of your application. Your application should check these variables after each GPIB call. The following sections describe each of these global variables and how you can use them in your application.

Status Word (ibsta)

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

`ibsta` is an integer-sized value. The least significant 16 bits of `ibsta` are meaningful. A bit value of one (1) indicates that a certain condition is in effect, and a bit value of zero (0) indicates that the condition is not in effect.

Table 3-1 shows the condition that each bit position represents and the bit mnemonics. For a detailed explanation of each of the status conditions, refer to Appendix B, *Status Word Conditions*.

Table 3-1. Status Word Layout

Mnemonic	Bit Pos.	Hex Value	Description
ERR	15	8000	GPIB error
TIMO	14	4000	Time limit exceeded
END	13	2000	END or EOS detected
SRQI	12	1000	SRQ interrupt received

Table 3-1. Status Word Layout (Continued)

Mnemonic	Bit Pos.	Hex Value	Description
CMPL	8	100	I/O completed
LOK	7	80	Lockout State
REM	6	40	Remote State
CIC	5	20	Controller-In-Charge
ATN	4	10	Attention is asserted
TACS	3	8	Talker
LACS	2	4	Listener
DTAS	1	2	Device Trigger State
DCAS	0	1	Device Clear State

The application header file `ugpib.h` included on your distribution medium defines each of the `ibsta` status bits. You can test for an `ibsta` status bit being set using the bitwise `and` operator (`&` in C/C++). For example, the `ibsta` ERR bit is bit 15 of `ibsta`. To check for a GPIB error, use the following statement after each GPIB call as shown:

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

Error Variable (iberr)

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

```
if (ibsta & ERR)
    printf("GPIB error %d encountered", iberr);
```

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

For more information on error codes and solutions, refer to the *Debugging Considerations* section of this chapter or Appendix C, *Error Codes and Solutions*.

Count Variables (ibcnt and ibcntl)

The count variables are updated after each read, write, or command function. `ibcnt` is an integer value and `ibcntl` is a long integer value. As implemented on most modern systems today, `ibcnt` and `ibcntl` are both 32-bit integers. On some older systems, such as MS-DOS, `ibcnt` is a 16-bit integer; on some newer systems, `ibcntl` is a 64-bit integer. For cross-platform compatibility, all applications should use `ibcntl`. If you are reading data, the count variables indicate the number of bytes read. If you are sending data or commands, the count variables reflect the number of bytes sent.

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

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

Compiling and Linking Your Application

To access the functions in the NI-488DDK driver from your application, you must link your application to the C language interface defined by the `cib.c` file. If you are using a user-level implementation of the DDK driver, in most cases you must also link your application to the low-level driver module itself, defined by the `ib.c` file.

The steps for compiling and linking your application program vary depending on your operating system and development environment. For example, the commands you might use to build and run an application on a UNIX-based system with a kernel-level driver are as follows:

```
cc my_application.c cib.c -o my_application
my_application
```

Alternatively, if many applications will be using the NI-488DDK driver on the sample UNIX system, you can compile the `cib.c` file separately and place it in a library for all applications to use with the following commands:

```
cc -c cib.c
ar r /usr/lib/libgpib.a cib.o
cc my_application.c -lgpib -o my_application
my_application
```

To access the routines in the sample IEEE 488.2 application library on the UNIX-based system, you would compile and link the `ni4882.c` file to your application as follows:

```
cc my_application.c ni4882.c -lgpib -o my_application
my_application
```

As with the standard `cib` module, if you desired to make the IEEE 488.2 library available to a number of applications, you could compile and archive the `ni4882.c` file in a library, rather than compiling the C source file with your application each time.

For specific instructions on the compiling and linking options available on your particular system, refer to the documentation that came with the system.

Debugging Considerations

This section contains typical errors you may encounter and some considerations for debugging your application.

Using the Global Status Variables

After each function call to your NI-488DDK 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 the *Checking Status with Global Variables* section of this chapter 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 B, *Status Word Conditions*, and Appendix C, *Error Codes and Solutions*. These appendices can help you interpret the state of the driver.

Configuration Errors

Some 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 temporarily reconfigure the driver while your application is running using the `ibeos` and `ibsad` functions.

Refer to the descriptions of these functions and others in Chapter 4, *NI-488DDK Functions*, for more information.

Timing Errors

In some cases, your application might fail because it is issuing the NI-488DDK 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 are controlled by the device and your application can adjust itself and work independently on any platform. Other delay mechanisms might cause varying delay times on different platforms.

Communication Errors

Repeat Addressing

Devices adhering to the IEEE 488.2 standard should remain in their current state until specific commands are sent across the GPIB to change their state. However, some devices require GPIB addressing before any GPIB activity. Therefore, you might need to make additional calls to `ibcmd` in your application to perform repeat addressing if your device does not remain in its currently addressed state.

Termination Method

You should learn the data termination method that your devices use. By default, your NI-488DDK software sends EOI on writes and terminates 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(ud,"COMMAND\x0D\x0A",9);
```

4

NI-488DDK Functions

This chapter lists the NI-488DDK functions and describes the purpose, format, input and output parameters, and possible errors for each function.

Legend

Function Names

The functions in this chapter are listed alphabetically.

Purpose

Each function description includes a brief statement of the purpose of the function.

Format

The format section describes the format of each function in the C programming language.

Input and Output

The input and output parameters for each function are listed. Function Return describes the return value of the function.

Description

The description section gives details about the purpose and effect of each function.

Examples

Some function descriptions include sample code showing how to use the function. For more detailed and complete examples, refer to the source code support files `ibchat.c` and `ni4882.c` that are included with your NI-488DDK software in the `util/` directory.

Possible Errors

Each function description includes a list of errors that could occur when it is invoked.

List of NI-488DDK Functions

Table 4-1 contains an alphabetical list of the NI-488DDK functions.

Table 4-1. NI-488DDK Functions

Function	Purpose
ibask	Return information about software configuration parameters
ibcac	Become Active Controller
ibcmd	Send GPIB commands
ibconfig	Change the software configuration parameters
ibdma	Enable or disable DMA
ibeos	Configure the end-of-string (EOS) termination mode or character
ibeot	Enable or disable the automatic assertion of the GPIB EOI line at the end of write I/O operations
ibfind	Open and initialize a GPIB board
ibgts	Go from Active Controller to Standby
ibist	Set or clear the board individual status bit for parallel polls
iblines	Return the status of the eight GPIB control lines
ibln	Check for the presence of a device on the bus
ibloc	Go to local
ibonl	Place the interface board online or offline
ibpad	Change the primary address
ibpoke	Change internal driver characteristics
ibppc	Parallel poll configure
ibrd	Read data into a user buffer
ibrpp	Conduct a parallel poll
ibrsc	Request or release system control
ibrsv	Request service and change the serial poll status byte
ibsad	Change or disable the secondary address
ibsic	Assert interface clear
ibsre	Set or clear the Remote Enable (REN) line
ibtmo	Change or disable the I/O timeout period
ibwait	Wait for GPIB events
ibwrt	Write data from a user buffer

IBASK

Purpose

Return information about software configuration parameters.

Format

```
int ibask (int ud, int option, int *value)
```

Note This function may not be available in all versions of NI-488DDK.

Input

`ud`	A board unit descriptor
`option`	Selects the configuration parameter

Output

`value`	Value of the configuration parameter
Function Return	The value of `ibsta`

Description

`ibask` returns the current value of various configuration parameters for the board. The current value of the selected configuration item is returned in the integer pointed to by `value`. Table 4-2 lists the valid configuration parameter options for `ibask`.

Table 4-2. ibask Board Configuration Parameter Options

Options (Constants)	Options (Values)	Returned Information
IbaHSCableLength	0x001F	0 = High-speed handshaking is disabled. 1 to 15 = The number of meters of GPIB cable in your system. The NI-488.2 software uses this information to select the appropriate timing in the high-speed handshaking mode.

Possible Errors

EARG	`option` is not a valid configuration parameter.
ECAP	`option` is not available in the NI-488DDK.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBCAC

Purpose

Become Active Controller.

Format

```
int ibcac (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	Determines if control is to be taken asynchronously or synchronously

Output

Function Return	The value of `ibsta`

Description

Using `ibcac`, the designated GPIB board attempts to become the Active Controller by asserting ATN. If `v` is zero, the GPIB board takes control asynchronously; if `v` is non-zero, the GPIB board takes control synchronously. Before you call `ibcac`, the GPIB board must already be CIC. To make the board CIC, use the `ibsic` function.

To take control synchronously, the GPIB board attempts to assert the ATN signal without corrupting transferred data. If this is not possible, the board takes control asynchronously.

To take control asynchronously, the GPIB board asserts ATN immediately without regard for any data transfer currently in progress.

Most applications do not need to use `ibcac`. Functions that require ATN to be asserted, such as `ibcmd`, do so automatically.

Possible Errors

EARG	`ud` is valid but does not refer to an interface board.
ECIC	The interface board is not Controller-In-Charge.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBCMD

Purpose

Send GPIB commands.

Format

```
int ibcmd (int ud, void *cmdbuf, long count)
```

Input

`ud`	A board unit descriptor
`cmdbuf`	Buffer of command bytes to send
`count`	Number of command bytes to send

Output

Function Return	The value of `ibsta`

Description

`ibcmd` sends `count` bytes from `cmdbuf` over the GPIB as command bytes (interface messages). The number of command bytes transferred is returned in the global variable, `ibcntl`. Refer to Table A-1, *Multiline Interface Messages*, for a list of the defined interface messages.

Command bytes configure the state of the GPIB, such as addressing devices to listen or talk.

Possible Errors

EABO	The timeout period expired before all of the command bytes were sent.
EARG	`ud` is valid but does not refer to an interface board.
ECIC	The interface board is not Controller-In-Charge.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.
ENOL	No Listeners are on the GPIB.

IBCONFIG

Purpose

Change the software configuration parameters.

Format

```
int ibconfig (int ud, int option, int value)
```

Input

`ud`	A board unit descriptor
`option`	Selects the configuration parameter
`value`	Value of the configuration parameter

Output

Function Return	The value of `ibsta`

Description

`ibconfig` alters the current value of the configuration item to the value for the selected board. `option` may be any of the defined constants in Table 4-3, and `value` must be valid for the parameter you are configuring. The previous setting of the configured item is returned in `iberr`.

Table 4-3. ibconfig Board Configuration Parameter Options

Options (Constants)	Options (Values)	Returned Information
IbcHSCableLength	0x001F	0 = High-speed handshaking is disabled. 1 to 15 = The number of meters of GPIB cable in your system. The NI-488.2 software uses this information to select the appropriate timing in the high-speed handshaking mode. (Default: 15)

Possible Errors

EARG	`option` is not a valid configuration parameter.
ECAP	`option` is not available in the NI-488DDK.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBDMA

Purpose

Enable or disable DMA.

Format

```
int ibdma (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	Enables or disables DMA

Output

Function Return	The value of `ibsta`

Description

`ibdma` enables or disables DMA operation. If `v` is non-zero, DMA transfers between the GPIB board and memory are used for read and write operations. If `v` is zero, programmed I/O is used.

The assignment made by this function remains in effect until `ibdma` is called again, the `ibonl` or `ibfind` function is called, or the system is restarted.

When `ibdma` is called and an error does not occur, the previous value of `v` is stored in `iberr`.

Possible Errors

ECAP	`option` is not available in the NI-488DDK.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBEOS

Purpose

Configure the end-of-string (EOS) termination mode or character.

Format

```
int ibeos (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	EOS mode and character information

Output

Function Return	The value of `ibsta`

Description

`ibeos` configures the EOS termination mode or EOS character for the board. The parameter `v` describes the new end-of-string (EOS) configuration to use. If `v` is zero, then the EOS configuration is disabled. Otherwise, the low byte is the EOS character and the upper byte contains flags which define the EOS mode.

Note Defining an EOS byte does not cause the driver to automatically send that byte at the end of write I/O. Your application is responsible for placing the EOS byte at the end of the data strings that it defines.

Table 4-4 describes the different EOS configurations and the corresponding values of `v`. If no error occurs during the call, the value of the previous EOS setting is returned in `iberr`.

Table 4-4. EOS Configurations

Bit	Configuration	Value of `v`	
		High Byte	**Low Byte**
A	Terminate read when EOS is detected.	00000100	EOS character
B	Set EOI with EOS on write function.	00001000	EOS character
C	Compare all 8 bits of EOS byte rather than low 7 bits (all read and write functions).	00010000	EOS character

IBEOS

(Continued)

Configuration bits A and C determine how to terminate read I/O operations. If bit A is set and bit C is clear, then a read ends when a byte that matches the low seven bits of the EOS character is received. If bits A and C are both set, then a read ends when a byte that matches all eight bits of the EOS character is received.

Configuration bits B and C determine when a write I/O operation asserts the GPIB EOI line. If bit B is set and bit C is clear, then EOI is asserted when the written character matches the low seven bits of the EOS character. If bits B and C are both set, then EOI is asserted when the written character matches all eight bits of the EOS character.

For more information on the termination of I/O operations, refer to Chapter 5, *GPIB Programming Techniques*.

Examples

```
ibeos (ud, 0x140A);    /* Configure the software to end reads on
                          newline character (hex 0A) for the unit
                          descriptor, ud */

ibeos (ud, 0x180A);    /* Configure the software to assert the GPIB
                          EOI line whenever the newline character
                          (hex 0A) is written out by the unit
                          descriptor, ud */
```

Possible Errors

EARG	The high byte of `v` contains invalid bits.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBEOT

Purpose

Enable or disable the automatic assertion of the GPIB EOI line at the end of write I/O operations.

Format

```
int ibeot (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	Enables or disables the end of transmission assertion of EOI

Output

Function Return	The value of `ibsta`

Description

`ibeot` enables or disables the assertion of the EOI line at the end of write I/O operations for the board `ud` describes. If `v` is non-zero, then EOI is asserted when the last byte of a GPIB write is sent. If `v` is zero, then nothing occurs when the last byte is sent. If no error occurs during the call, then the previous value of EOT is returned in `iberr`.

For more information on the termination of I/O operations, refer to Chapter 5, *GPIB Programming Techniques*.

Possible Errors

EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBFIND

Purpose

Open and initialize a GPIB board descriptor.

Format

```
int ibfind (char *udname)
```

Input

`udname`	A GPIB board name

Output

Function Return	The board descriptor, or **`-1`**

Description

`ibfind` acquires a descriptor for a GPIB board; this board descriptor can be used in subsequent NI-488DDK functions.

`ibfind` performs the equivalent of an `ibonl 1` to initialize the board descriptor. The unit descriptor that `ibfind` returns remains valid until you use `ibonl 0` to put the board offline.

If `ibfind` is unable to get a valid descriptor, **`-1`** is returned; the ERR bit is set in `ibsta` and `iberr` contains EDVR.

Possible Errors

EDVR	Either `udname` is not recognized as a board name or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBGTS

Purpose

Go from Active Controller to Standby.

Format

```
int ibgts (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	Determines whether to perform acceptor handshaking

Output

Function Return	The value of `ibsta`

Description

`ibgts` causes the GPIB board at `ud` to go to Standby Controller and the GPIB ATN line to be unasserted. If `v` is non-zero, acceptor handshaking or shadow handshaking is performed until END occurs or until ATN is reasserted by a subsequent `ibcac` call. With this option, the GPIB board can participate in data handshake as an acceptor without actually reading data. If END is detected, the interface board enters a Not Ready For Data (NRFD) handshake holdoff state which results in hold off of subsequent GPIB transfers. If `v` is 0, no acceptor handshaking or holdoff is performed.

Before performing an `ibgts` with shadow handshake, call the `ibeos` function to establish proper EOS modes.

For details on the IEEE-488.1 handshake protocol, refer to the ANSI/IEEE Standard 488.1-1987 document.

Possible Errors

EADR	`v` is non-zero, and either ATN is low or the interface board is a Talker or Listener.
EARG	`ud` is valid but does not refer to an interface board.
ECIC	The interface board is not Controller-In-Charge.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBIST

Purpose

Set or clear the board individual status bit for parallel polls.

Format

```
int ibist (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	Indicates whether to set or clear the `ist` bit

Output

Function Return	The value of `ibsta`

Description

`ibist` sets the interface board `ist` (individual status) bit according to `v`. If `v` is zero, the `ist` bit is cleared; if `v` is non-zero, the `ist` bit is set. The previous value of the `ist` bit is returned in `iberr`.

For more information on parallel polling, refer to Chapter 5, *GPIB Programming Techniques*.

Possible Errors

EARG	`ud` is valid but does not refer to an interface board.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBLINES

Purpose

Return the status of the eight GPIB control lines.

Format

```
int iblines (int ud, short *clines)
```

Input

`ud`	A board unit descriptor

Output

`clines`	Returns GPIB control line state information
Function Return	The value of `ibsta`

Description

`iblines` returns the state of the GPIB control lines in `clines`. The low-order byte (bits 0 through 7) of `clines` contains a mask indicating the capability of the GPIB interface board to sense the status of each GPIB control line. The upper byte (bits 8 through 15) contains the GPIB control line state information. The following is a pattern of each byte.

7	6	5	4	3	2	1	0
EOI	ATN	SRQ	REN	IFC	NRFD	NDAC	DAV

To determine if a GPIB control line is asserted, first check the appropriate bit in the lower byte to determine if the line can be monitored. If the line can be monitored (indicated by a 1 in the appropriate bit position), then check the corresponding bit in the upper byte. If the bit is set (1), the corresponding control line is asserted. If the bit is clear (0), the control line is unasserted.

IBLINES

(Continued)

Example

```
short lines;
iblines (ud, &lines);
if (lines & ValidREN) {  /* check to see if REN is asserted */
   if (lines & BusREN) {
      printf ("REN is asserted");
   }
}
```

Possible Errors

EARG	`ud` is valid but does not refer to an interface board.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBLN

Purpose

Check for the presence of a device on the bus.

Format

```
int ibln (int ud, int pad, int sad, short *listen)
```

Input

`ud`	A board unit descriptor
`pad`	The primary GPIB address of the device
`sad`	The secondary GPIB address of the device

Output

`listen`	Indicates if a device is present or not
Function Return	The value of `ibsta`

Description

`ibln` determines whether there is a listening device at the GPIB address designated by the `pad` and `sad` parameters. If a Listener is detected, a non-zero value is returned in `listen`. If no Listener is found, zero is returned.

The `pad` parameter can be any valid primary address (a value between 0 and 30). The `sad` parameter can be any valid secondary address (a value between 96 to 126), or one of the constants `NO_SAD` or `ALL_SAD`. The constant `NO_SAD` designates that no secondary address is to be tested (only a primary address is tested). The constant `ALL_SAD` designates that all secondary addresses are to be tested.

Possible Errors

EARG	Either the `pad` or `sad` argument is invalid.
ECIC	The interface board is not Controller-In-Charge.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBLOC

Purpose

Go to Local.

Format

```
int ibloc (int ud)
```

Input

`ud`	A board unit descriptor

Output

Function Return	The value of `ibsta`

Description

`ibloc` places the board in local mode if it is not in a lockout state. The board is in a lockout state if LOK appears in the status word `ibsta`. If the board is in a lockout state, the call has no effect.

The `ibloc` function is used to simulate a front panel RTL (Return to Local) switch if the computer is used as an instrument.

Possible Errors

EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBONL

Purpose

Place the interface board online or offline.

Format

```
int ibonl (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	Indicates whether the board is to be taken online or offline

Output

Function Return	The value of `ibsta`

Description

`ibonl` resets the board and places all its software configuration parameters in their pre-configured state. In addition, if `v` is zero, the interface board is taken offline. If `v` is non-zero, the interface board is left operational, or online.

If an interface board is taken offline, the board descriptor (`ud`) is no longer valid. You must execute an `ibfind` to access the board again.

Possible Errors

EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBPAD

Purpose

Change the primary address.

Format

```
int ibpad (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	GPIB primary address

Output

Function Return	The value of `ibsta`

Description

`ibpad` sets the primary GPIB address of the board to `v`, an integer ranging from 0 to 30. If no error occurs during the call, then `iberr` contains the previous GPIB primary address.

Possible Errors

EARG	`v` is not a valid primary GPIB address; it must be in the range 0 to 30.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBPOKE

Purpose

Change internal driver characteristics.

Format

```
int ibpoke (int ud, int option, int v)
```

Input

`ud`	A board unit descriptor
`option`	A parameter that selects the characteristic to be changed
`v`	The value to which the selected characteristic is to be changed

Output

Function Return	The value of `ibsta`

Description

`ibpoke` modifies miscellaneous internal characteristics within the NI-488DDK driver, such as turning on or off certain types of debugging statements. The operations that can be performed with `ibpoke` can vary with different NI-488 drivers. This function is intended for driver developer use only and should generally not be used in end-user application development. For these reasons, `ibpoke` is not documented in standard NI-488 manual sets.

For the specific options and values supported by a particular version of driver, refer to the source code for `ibpoke` in the files `ni488.c` and `ni_suprt.c`.

Examples

```
ibpoke (ud, 1, 1);    /* Turn all driver debugging statements ON */
ibpoke (ud, 1, 0);    /* Turn all driver debugging statements OFF */
```

Possible Errors

EARG	Either `option` or `v` is invalid.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBPPC

Purpose

Parallel poll configure.

Format

```
int ibppc (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	Parallel poll enable/disable (PPE/PPD) value

Output

Function Return	The value of `ibsta`

Description

`ibppc` performs a local parallel poll configuration on the interface board using the parallel poll configuration value `v`. Valid parallel poll messages are 96 to 126 (hex 60 to hex 7E) or zero to send PPD. If no error occurs during the call, then `iberr` contains the previous value of the local parallel poll configuration.

For more information on parallel polling, refer to Chapter 5, *GPIB Programming Techniques*.

Possible Errors

EARG	`v` does not contain a valid parallel poll enable (PPE) or parallel poll disable (PPD) message.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBRD

Purpose

Read data into a user buffer.

Format

```
int ibrd (int ud, void *rdbuf, long count)
```

Input

`ud`	A board unit descriptor
`count`	Number of bytes to be read from the GPIB

Output

`rdbuf`	Address of buffer into which data is read
Function Return	The value of `ibsta`

Description

`ibrd` reads up to `count` bytes of data and places the data into the buffer specified by `rdbuf`. `ibrd` assumes that the GPIB is already properly addressed. The operation terminates normally when `count` bytes have been received or END is received. The operation terminates with an error if the transfer could not complete within the timeout period or, if the board is not CIC, the CIC sends a Device clear on the GPIB. The actual number of bytes transferred is returned in the global variable `ibcntl`.

Possible Errors

EABO	Either `count` bytes or END was not received within the timeout period or a Device Clear message was received after the read operation began.
EADR	The GPIB is not correctly addressed; use `ibcmd` to address the GPIB.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBRPP

Purpose

Conduct a parallel poll.

Format

```
int ibrpp (int ud, char *ppr)
```

Input

`ud`	A board unit descriptor

Output

`ppr`	Parallel poll response byte
Function Return	The value of `ibsta`

Description

`ibrpp` parallel polls all the devices on the GPIB. The result of this poll is returned in `ppr`.

For more information on parallel polling, refer to Chapter 5, *GPIB Programming Techniques*.

Possible Errors

ECIC	The interface board is not Controller-In-Charge.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBRSC

Purpose

Request or release system control.

Format

```
int ibrsc (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	Determines if system control is to be requested or released

Output

Function Return	The value of `ibsta`

Description

`ibrsc` requests or releases the capability to send Interface Clear (IFC) and Remote Enable (REN) messages. If `v` is zero, the board releases system control, and functions requiring System Controller capability are not allowed. If `v` is non-zero, functions requiring System Controller capability are subsequently allowed. If no error occurs during the call, then `iberr` contains the previous System Controller state of the board.

Possible Errors

EARG	`ud` is a valid descriptor but does not refer to a board.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBRSV

Purpose

Request service and change the serial poll status byte.

Format

```
int ibrsv (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	Serial poll status byte

Output

Function Return	The value of `ibsta`

Description

`ibrsv` is used to request service from the Controller and to provide the Controller with an application-dependent status byte when the Controller serial polls the GPIB board.

The value `v` is the status byte that the GPIB board returns when serial polled by the Controller-In-Charge. If bit 6 (hex 40) is set in `v`, the GPIB board requests service from the Controller by asserting the GPIB SRQ line. When `ibrsv` is called and an error does not occur, the previous status byte is returned in `iberr`.

Possible Errors

EARG	`ud` is a valid descriptor but does not refer to a board.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBSAD

Purpose

Change or disable the secondary address.

Format

```
int ibsad (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	GPIB secondary address

Output

Function Return	The value of `ibsta`

Description

`ibsad` changes the secondary GPIB address of the given board to `v`, an integer in the range 96 to 126 (hex 60 to hex 7E) or zero. If `v` is zero, secondary addressing is disabled. If no error occurs during the call, then the previous value of the GPIB secondary address is returned in `iberr`.

Possible Errors

EARG	`v` is non-zero and outside the legal range 96 to 126.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.

IBSIC

Purpose

Assert interface clear.

Format

```
int ibsic (int ud)
```

Input

`ud`	A board unit descriptor

Output

Function Return	The value of `ibsta`

Description

`ibsic` asserts the GPIB interface clear (IFC) line for at least 100 µs if the GPIB board is System Controller. This initializes the GPIB and makes the interface board CIC and Active Controller with ATN asserted.

The IFC signal resets only the GPIB interface functions of bus devices and not the internal device functions. Consult your device documentation to determine how to reset the internal functions of your device.

Possible Errors

EARG	`ud` is a valid descriptor but does not refer to a board.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.
ESAC	The board does not have System Controller capability.

IBSRE

Purpose

Set or clear the Remote Enable line.

Format

```
int ibsre (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	Indicates whether to set or clear the REN line

Output

Function Return	The value of `ibsta`

Description

If `v` is non-zero, the GPIB Remote Enable (REN) line is asserted. If `v` is zero, REN is unasserted. The previous value of REN is returned in `iberr`.

Devices use REN to choose between local and remote modes of operation. A device should not actually enter remote mode until it receives its listen address.

Possible Errors

EARG	`ud` is a valid descriptor but does not refer to a board.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.
ESAC	The board does not have System Controller capability.

IBTMO

Purpose

Change or disable the timeout period.

Format

```
int ibtmo (int ud, int v)
```

Input

`ud`	A board unit descriptor
`v`	Timeout duration code

Output

Function Return	The value of `ibsta`

Description

`ibtmo` sets the timeout period of the board to `v`. The timeout period is used to select the maximum duration allowed for an I/O operation (for example, `ibrd` and `ibwrt`) or for an `ibwait` operation with TIMO in the wait mask. If the operation does not complete before the timeout period elapses, then the operation is aborted and TIMO is returned in `ibsta`. Refer to Table 4-5 for a list of valid timeout values. These timeout values represent the minimum timeout period. The actual period may be longer.

IBTMO

(Continued)

Table 4-5. Timeout Code Values

Constant	Value of v	Minimum Timeout
TNONE	0	Disabled/no timeout
T10us	1	10 µs
T30us	2	30 µs
T100us	3	100 µs
T300us	4	300 µs
T1ms	5	1 ms
T3ms	6	3 ms
T10ms	7	10 ms
T30ms	8	30 ms
T100ms	9	100 ms
T300ms	10	300 ms
T1s	11	1 s
T3s	12	3 s
T10s	13	10 s
T30s	14	30 s
T100s	15	100 s
T300s	16	300 s
T1000s	17	1,000 s

Possible Errors

EARG v is invalid.
EDVR Either ud is invalid or the NI-488DDK driver is not installed.
ENEB The interface board is not installed or is not properly configured.

IBWAIT

Purpose

Wait for GPIB events.

Format

`int ibwait (int ud, int mask)`

Input

`ud`	A board unit descriptor
`mask`	Bit mask of GPIB events to wait for

Output

Function Return	The value of `ibsta`

Description

`ibwait` monitors the events that `mask` specifies and delays processing until one or more of the events occurs. If the wait mask is zero, `ibwait` returns immediately with the updated `ibsta` status word. If TIMO is set in the wait mask, `ibwait` returns when the timeout period has elapsed, if one or more of the other specified events have not already occurred. If TIMO is not set in the wait mask, then `ibwait` waits indefinitely for one or more of the specified events to occur. The existing `ibwait` mask bits are identical to the `ibsta` bits, and they are described in Table 4-6. You can configure the timeout period using the `ibtmo` function.

IBWAIT

(Continued)

Table 4-6. Wait Mask Layout

Mnemonic	Bit Pos.	Hex Value	Description
TIMO	14	4000	Use the timeout period (see `ibtmo`) to limit the wait period
END	13	2000	END or EOS is detected
SRQI	12	1000	SRQ is asserted
CMPL	8	100	I/O completed
LOK	7	80	GPIB board is in Lockout State
REM	6	40	GPIB board is in Remote State
CIC	5	20	GPIB board is CIC
ATN	4	10	Attention is asserted
TACS	3	8	GPIB board is Talker
LACS	2	4	GPIB board is Listener
DTAS	1	2	GPIB board is in Device Trigger State
DCAS	0	1	GPIB board is in Device Clear State

Possible Errors

EARG The bit set in `mask` is invalid.
EDVR Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB The interface board is not installed or is not properly configured.

IBWRT

Purpose

Write data from a user buffer.

Format

```
int ibwrt (int ud, void *wrtbuf, long count)
```

Input

`ud`	A board unit descriptor
`wrtbuf`	Address of the buffer containing the bytes to write
`count`	Number of bytes to be written

Output

Function Return	The value of `ibsta`

Description

`ibwrt` writes `count` bytes of data from the buffer specified by `wrtbuf`; `ibwrt` assumes that the GPIB is already properly addressed. The operation terminates normally when `count` bytes have been sent. The operation terminates with an error if `count` bytes could not be sent within the timeout period or, if the board is not CIC, the CIC sends Device Clear on the GPIB. The actual number of bytes transferred is returned in the global variable `ibcntl`.

Possible Errors

EABO	Either `count` bytes were not sent within the timeout period, or a Device Clear message was received after the write operation began.
EADR	The GPIB is not correctly addressed; use `ibcmd` to address the GPIB.
EDVR	Either `ud` is invalid or the NI-488DDK driver is not installed.
ENEB	The interface board is not installed or is not properly configured.
ENOL	No Listeners were detected on the bus.

5

GPIB Programming Techniques

This chapter describes techniques for using some NI-488DDK functions in your application.

For more detailed information about each function, refer to Chapter 4, *NI-488DDK Functions*.

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-488DDK 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-488DDK driver asserts the GPIB EOI line when the last byte of a write is sent out on the GPIB. If it is disabled, the EOI line is *not* asserted with the last byte of a write.

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

- A 7-bit or 8-bit EOS byte
- EOS comparison method—This indicates whether the EOS byte has seven or eight significant bits. For a 7-bit EOS byte, the eighth bit of the EOS byte is ignored.
- EOS write method—If you enable this, the NI-488DDK driver automatically asserts the GPIB EOI line when the EOS byte is written to the GPIB. If the buffer passed into an `ibwrt` call contains five occurrences of the EOS byte, the EOI line is asserted as each of the five EOS bytes are written to the GPIB. If an `ibwrt` buffer does not contain an occurrence of the EOS byte, the EOI line is not asserted (unless the EOT mode is enabled, in which case the EOI line is asserted with the last byte of the write).
- EOS read method—If you enable this, the NI-488DDK driver terminates `ibrd` calls when the EOS byte is detected on the GPIB or

when the GPIB EOI line is asserted or when the specified count is reached. If you disable the EOS read method, `ibrd` calls terminate only when the GPIB EOI line is asserted or the specified count has been read.

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.

Talker/Listener Applications

Although designed for Controller-In-Charge applications, you can also use the NI-488DDK software in most non-Controller situations. These situations are known as Talker/Listener applications because the interface board is not the GPIB Controller.

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

Serial Polling

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

Service Requests from IEEE 488 Devices

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

Service Requests from IEEE 488.2 Devices

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

SRQ and Serial Polling with NI-488DDK Functions

The 488.2 application library included with the NI-488DDK driver contains some high-level routines that you can use to conduct SRQ servicing and serial polling. Routines pertinent to SRQ servicing and serial polling are `ni4882_ReadStatusByte`, `ni4882_TestSRQ`, and `ni4882_WaitSRQ`.

`ni4882_ReadStatusByte` serial polls a single device and returns its status byte.

`ni4882_TestSRQ` determines whether the SRQ line is asserted or unasserted, and returns to the caller immediately.

`ni4882_WaitSRQ` is similar to `ni4882_TestSRQ`, except that `ni4882_WaitSRQ` suspends the application until either SRQ is asserted or the timeout period is exceeded.

You can also use the IEEE 488.2 routines mentioned in this section to construct your own SRQ servicing routines using the low-level functions of the NI-488DDK driver. Refer to the file `ni4882.c` for more information.

Parallel Polling

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

Implementing a Parallel Poll with NI-488DDK Functions

Complete the following steps to implement parallel polling using NI-488DDK functions. Each step contains example code.

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

   Parallel poll configuration 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 at primary address 3 for parallel polling using NI-488DDK functions. The device asserts DIO7 if its `ist` bit = 0, therefore, 0110 0110 (hex 66) is the parallel poll configuration byte.

   ```
   #include "ugpib.h"
   char ppr;
   ud = ibfind("gpib0");
   ```

```
ibsic(ud);
ibcmd(ud, "?#\x05\x66", 4);
```

If the GPIB interface board configures itself for a parallel poll, you should use the `ibppc` function. Pass the 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(ud, 0x6C);
ibist(ud, 1);
```

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

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

3. Unconfigure the device for parallel polling with `ibcmd`. 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.

```
ibcmd(ud,"?#\x05\x70", 4);
```

A

Multiline Interface Messages

This appendix contains a multiline interface message reference list, which describes the mnemonics and messages that correspond to the interface functions. These multiline interface messages are sent and received with ATN asserted.

For more information about these messages, refer to the ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation*.

Table A-1. Multiline Interface Messages

Hex	Dec	ASCII	Msg
00	0	NUL	—
01	1	SOH	GTL
02	2	STX	—
03	3	ETX	—
04	4	EOT	SDC
05	5	ENQ	PPC
06	6	ACK	—
07	7	BEL	—
08	8	BS	GET
09	9	HT	TCT
0A	10	LF	—
0B	11	VT	—
0C	12	FF	—
0D	13	CR	—
0E	14	SO	—
0F	15	SI	—
10	16	DLE	—
11	17	DC1	LLO
12	18	DC2	—
13	19	DC3	—
14	20	DC4	DCL
15	21	NAK	PPU
16	22	SYN	—
17	23	ETB	—
18	24	CAN	SPE
19	25	EM	SPD
1A	26	SUB	—
1B	27	ESC	—
1C	28	FS	—
1D	29	GS	—
1E	30	RS	—
1F	31	US	CFE

Hex	Dec	ASCII	Msg
20	32	SP	MLA0
21	33	!	MLA1
22	34	"	MLA2
23	35	#	MLA3
24	36	$	MLA4
25	37	%	MLA5
26	38	&	MLA6
27	39	'	MLA7
28	40	(	MLA8
29	41	)	MLA9
2A	42	*	MLA10
2B	43	+	MLA11
2C	44	,	MLA12
2D	45	-	MLA13
2E	46	.	MLA14
2F	47	/	MLA15
30	48	0	MLA16
31	49	1	MLA17
32	50	2	MLA18
33	51	3	MLA19
34	52	4	MLA20
35	53	5	MLA21
36	54	6	MLA22
37	55	7	MLA23
38	56	8	MLA24
39	57	9	MLA25
3A	58	:	MLA26
3B	59	;	MLA27
3C	60	<	MLA28
3D	61	=	MLA29
3E	62	>	MLA30
3F	63	?	UNL

Multiline Interface Message Definitions

CFE†	Configuration Enable	GTL	Go To Local
CFG†	Configure	LLO	Local Lockout
DCL	Device Clear	MLA	My Listen Address
GET	Group Execute Trigger	MSA	My Secondary Address

†This multiline interface message is a proposed extension to the IEEE 488.1 specification to support the HS488 high-speed protocol.

Table A-1. Multiline Interface Messages (Continued)

Hex	Dec	ASCII	Msg	Hex	Dec	ASCII	Msg
40	64	@	MTA0	60	96	`	MSA0, PPE
41	65	A	MTA1	61	97	a	MSA1, PPE, CFG1
42	66	B	MTA2	62	98	b	MSA2, PPE, CFG2
43	67	C	MTA3	63	99	c	MSA3, PPE, CFG3
44	68	D	MTA4	64	100	d	MSA4, PPE, CFG4
45	69	E	MTA5	65	101	e	MSA5, PPE, CFG5
46	70	F	MTA6	66	102	f	MSA6, PPE, CFG6
47	71	G	MTA7	67	103	g	MSA7, PPE, CFG7
48	72	H	MTA8	68	104	h	MSA8, PPE, CFG8
49	73	I	MTA9	69	105	i	MSA9, PPE, CFG9
4A	74	J	MTA10	6A	106	j	MSA10, PPE, CFG10
4B	75	K	MTA11	6B	107	k	MSA11, PPE, CFG11
4C	76	L	MTA12	6C	108	l	MSA12, PPE, CFG12
4D	77	M	MTA13	6D	109	m	MSA13, PPE, CFG13
4E	78	N	MTA14	6E	110	n	MSA14, PPE, CFG14
4F	79	O	MTA15	6F	111	o	MSA15, PPE, CFG15
50	80	P	MTA16	70	112	p	MSA16, PPD
51	81	Q	MTA17	71	113	q	MSA17, PPD
52	82	R	MTA18	72	114	r	MSA18, PPD
53	83	S	MTA19	73	115	s	MSA19, PPD
54	84	T	MTA20	74	116	t	MSA20, PPD
55	85	U	MTA21	75	117	u	MSA21, PPD
56	86	V	MTA22	76	118	v	MSA22, PPD
57	87	W	MTA23	77	119	w	MSA23, PPD
58	88	X	MTA24	78	120	x	MSA24, PPD
59	89	Y	MTA25	79	121	y	MSA25, PPD
5A	90	Z	MTA26	7A	122	z	MSA26, PPD
5B	91	[	MTA27	7B	123	{	MSA27, PPD
5C	92	\	MTA28	7C	124	|	MSA28, PPD
5D	93	]	MTA29	7D	125	}	MSA29, PPD
5E	94	^	MTA30	7E	126	~	MSA30, PPD
5F	95	_	UNT	7F	127	DEL	—

Multiline Interface Message Definitions (Continued)

MTA My Talk Address
PPC Parallel Poll Configure
PPD Parallel Poll Disable
PPE Parallel Poll Enable
PPU Parallel Poll Unconfigure
SDC Selected Device Clear
SPD Serial Poll Disable
SPE Serial Poll Enable
TCT Take Control
UNL Unlisten
UNT Untalk

B

Status Word Conditions

This appendix describes the conditions reported in the status word, `ibsta`.

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

Table B-1 shows the status word layout.

Table B-1. Status Word Layout

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

ERR

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 C, *Error Codes and Solutions*, describes error codes that are recorded in `iberr` along with possible solutions. ERR is cleared following any call that does not result in an error.

TIMO

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 mask parameter is set and the time limit expires. TIMO is also set following any I/O functions (for example, `ibcmd`, `ibrd`, and `ibwrt`) if a timeout occurs during one of these calls. TIMO is cleared in all other circumstances.

END

END indicates either that the GPIB EOI line has been asserted or, if you configure the software to terminate a read on an EOS byte, that the EOS byte has been received. 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.

SRQI

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

CMPL

CMPL indicates the condition of I/O operations. Because I/O calls in the NI-488DDK driver are all synchronous (meaning the call does not return until the operation is complete), CMPL is always set.

LOK

LOK indicates whether the board is in a lockout state. While LOK is set, the `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

REM indicates whether 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 you call the `ibloc` function while the LOK bit is cleared in the status word

CIC

CIC indicates whether the GPIB board is the Controller-In-Charge. CIC is set either when you execute the `ibsic` function 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

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

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

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

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

DCAS indicates whether the GPIB board has detected a device clear command. DCAS is set whenever the GPIB board detects that the Device Clear (DCL) command has been sent by another Controller, or whenever the GPIB board as a Listener detects that the Selected Device Clear (SDC) command has been sent by another Controller.

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

C

Error Codes and Solutions

This appendix describes each error, including conditions under which it might occur and possible solutions.

Table C-1 lists the GPIB error codes.

Table C-1. GPIB Error Codes

Error Mnemonic	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
ECAP	11	No capability for operation

EDVR (0)

EDVR is returned when the board name passed to `ibfind` cannot be accessed. The global variable `ibcntl` contains an error code. This error occurs when you try to access a board that is not installed or configured properly.

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

Solutions

Following are some possible solutions:

- Use only board names configured in the driver source code as parameters to the `ibfind` function.
- Use the unit descriptor returned from `ibfind` as the first parameter in subsequent NI-488DDK functions. Examine the variable before the failing function to make sure its value has not been corrupted.

ECIC (1)

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

- Any board-level NI-488DDK functions that issue GPIB command bytes: `ibcmd`, `ibln`, and `ibrpp`
- `ibcac` and `ibgts`

Solutions

Following are some possible solutions:

- Use `ibsic` 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 you attempt a write operation without addressing Listeners. ENOL can also indicate that the GPIB address the application uses for a device does not match the GPIB address of any device connected to the bus, that the GPIB cable is not connected to the device, or that the device is not powered on.

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

Solutions

Following are some possible solutions:

- Make sure that the GPIB address you are using 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.
- 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.

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

Following are some possible solutions:

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

EARG (4)

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

- `ibtmo` called with a value not in the range 0 through 17
- `ibeos` called with meaningless bits set in the high byte of the second parameter
- `ibpad` or `ibsad` called with invalid addresses
- `ibppc` called with invalid parallel poll configurations

Solution

Make sure that the parameters passed to the NI-488DDK function are valid.

ESAC (5)

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

Solution

Give the GPIB board System Controller capability by calling `ibrsc 1`.

EABO (6)

EABO indicates that an I/O operation has been canceled, usually due to a timeout condition. Another cause is 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

Following are some possible solutions:

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

ENEB (7)

ENEB occurs when no GPIB board exists at the I/O address specified when the driver is installed. This problem happens when the board is not physically plugged into the system, the I/O address specified during configuration does not match the actual board setting, or there is a system conflict with the base I/O address.

Solution

Make sure there is a GPIB board in your computer that is properly configured both in hardware and software using a valid base I/O address.

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.

Solution

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

D

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at `ni.com` for technical support and professional services:

- **Support**—Online technical support resources include the following:
  - **Self-Help Resources**—For immediate answers and solutions, visit our extensive library of technical support resources available in English, Japanese, and Spanish at `ni.com/support`. These resources are available for most products at no cost to registered users and include software drivers and updates, a KnowledgeBase, product manuals, step-by-step troubleshooting wizards, hardware schematics and conformity documentation, example code, tutorials and application notes, instrument drivers, discussion forums, a measurement glossary, and so on.
  - **Assisted Support Options**—Contact NI engineers and other measurement and automation professionals by visiting `ni.com/ask`. Our online system helps you define your question and connects you to the experts by phone, discussion forum, or email.
- **Training**—Visit `ni.com/custed` for self-paced tutorials, videos, and interactive CDs. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, NI Alliance Program members can help. To learn more, call your local NI office or visit `ni.com/alliance`.

If you searched `ni.com` and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of `ni.com/niglobal` to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Glossary

Prefix	Meaning	Value
p-	pico-	10^{-12}
μ-	micro-	10^{-6}
m-	milli-	10^{-3}
k-	kilo-	10^{3}
M-	mega-	10^{6}

A

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

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

ANSI — American National Standards Institute.

API — Application programming interface.

ASCII — American Standard Code for Information Interchange.

asynchronous — An action or event that occurs at an unpredictable time with respect to the execution of a program.

B

base I/O address — *See* I/O address.

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

C

caller — A place in the program from which a call is made; the calling function.

CFE	Configuration Enable. The GPIB command which precedes CFGn and is used to place devices into their configuration mode.
CFGn	These GPIB commands (CFG1 through CFG15) follow CFE and are used to configure all devices for the number of meters of cable in the system so that HS488 transfers occur without errors.
CIC	Controller-In-Charge. The device that manages the GPIB by sending interface messages to other devices.
CPU	Central processing unit.

D

DAV	Data Valid. One of the three GPIB handshake lines. *See* handshake.
DCL	Device Clear. The GPIB command used to reset the device or internal functions of all devices. *See* SDC.
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.
driver	Device driver software installed within the operating system.

E

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

F

Function Return	Describes the return value of the function.

G

GET	Group Execute Trigger. It is the GPIB command used to trigger a device or internal function of an addressed Listener.
GPIB	General Purpose Interface Bus is the common name for the communications interface system defined in ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1992.
GPIB address	The address of a device on the GPIB, composed of a primary address (MLA and MTA) and perhaps a secondary address (MSA). The GPIB board has both a GPIB address and an I/O address.
GPIB board	Refers to the National Instruments family of GPIB interface boards.
GTL	Go To Local. It is the GPIB command used to place an addressed Listener in local (front panel) control mode.

H

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

I

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.
`ibcnt`	After each NI-488 I/O function, this global variable contains the actual number of bytes transmitted.
`iberr`	A global variable that contains the specific error code associated with a function call that failed.
`ibsta`	At the end of each function call, this global variable (status word) contains status information.
IEEE	Institute of Electrical and Electronic Engineers.
interface message	A broadcast message sent from the Controller to all devices and used to manage the GPIB. Interface messages are also referred to as GPIB commands.
`ist`	An Individual Status bit of the status byte used in the Parallel Poll Configure function.

K

kernel	The set of programs in an operating system that implements basic system functions.
kernel-level implementation	The linking or installation of the NI-488DDK driver into the operating system kernel so that the driver functions as a general system resource available to all application programs.

L

language interface	Code that enables an application program that uses NI-488DDK functions to access the driver.

Listener	A GPIB device that receives data messages from a Talker.
LLO	Local Lockout. The GPIB command used to tell all devices that they may or should ignore remote (GPIB) data messages or local (front panel) controls, depending on whether the device is in local or remote program mode.

M

m	Meters.
MAV	The Message Available bit is part of the IEEE 488.2-defined status byte which is received from a device responding to a serial poll.
MLA	My Listen Address. A GPIB command used to address a device to be a Listener. It can be any one of the 31 primary addresses.
MSA	My Secondary Address. The GPIB command used to address a device to be a Listener or a Talker when extended (two byte) addressing is used. The complete address is a MLA or MTA address followed by an MSA address. There are 31 secondary addresses for a total of 961 distinct listen or talk addresses for devices.
MTA	My Talk Address. A GPIB command used to address a device to be a Talker. It can be any one of the 31 primary addresses.

N

NDAC	Not Data Accepted. One of the three GPIB handshake lines. *See* handshake.
NRFD	Not Ready For Data. One of the three GPIB handshake lines. *See* handshake.

O

OS	Operating system.

P

parallel poll	The process of polling all configured devices at once and reading a composite poll response. *See* serial poll.
PPC	Parallel Poll Configure. It is the GPIB command used to configure an addressed Listener to participate in polls.
PPD	Parallel Poll Disable. It is the GPIB command used to disable a configured device from participating in polls. There are 16 PPD commands.
PPE	Parallel Poll Enable. It is the GPIB command used to enable a configured device to participate in polls and to assign a DIO response line. There are 16 PPE commands.
PPU	Parallel Poll Unconfigure. It is the GPIB command used to disable any device from participating in polls.

S

s	Seconds.
SDC	Selected Device Clear. The GPIB command used to reset internal or device functions of an addressed Listener. *See* DCL.
serial poll	The process of polling and reading the status byte of one device at a time. *See* parallel poll.
service request	*See* SRQ.
SPD	Serial Poll Disable. The GPIB command used to cancel an SPE command.
SPE	Serial Poll Enable. The GPIB command used to enable a specific device to be polled. That device must also be addressed to talk. *See* SPD.
SRQ	Service Request. The GPIB line that a device asserts to notify the CIC that the device needs servicing.
status byte	The IEEE 488.2-defined data byte sent by a device when it is serially polled.
status word	*See* `ibsta`.

synchronous	Refers to the relationship between the NI-488DDK driver functions and a process when executing driver functions is predictable; the process is blocked until the driver completes the function.
System Controller	The single designated Controller that can assert control (become CIC of the GPIB) by sending the Interface Clear (IFC) message. Other devices can become CIC only by having control passed to them.

T

Talker	A GPIB device that sends data messages to Listeners.
TCT	Take Control. The GPIB command used to pass control of the bus from the current Controller to an addressed Talker.
timeout	A feature of the NI-488DDK driver that prevents I/O functions from hanging indefinitely when there is a problem on the GPIB.

U

`ud`	Unit descriptor. A variable name and first argument of each function call that contains the unit descriptor of the GPIB interface board or other GPIB device that is the object of the function.
UNL	Unlisten. The GPIB command used to unaddress any active Listeners.
UNT	Untalk. The GPIB command used to unaddress an active Talker.
user-level implementation	The static or dynamic linking of the NI-488DDK driver directly to a user application program. This implementation method is not available on some operating systems, for which a kernel-level implementation is the only option.

Index

E

F

G

H

I

K

L

M

N

O

P

R

S

T

W