NI-488.2M™
Software Reference Manual
for OS/2

January 1995 Edition

Part Number 370950A-01

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

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

Limited Warranty

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

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

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

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

installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Warning Regarding Medical and Clinical Use of National Instruments Products

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

Contents

Contents

Contents

Figures

Tables

About This Manual

This manual describes the features and functions of the NI-488.2M software for OS/2. The NI-488.2M software package is meant to be used with OS/2 (IBM Operating System/2) version 2.0 or higher. This manual assumes that you are already familiar with the OS/2 system.

Organization of This Manual

This manual is organized as follows:

- Chapter 1, *NI-488.2M Software Description*, describes the NI-488.2M software package and explains how the software works with your OS/2 system.

- Chapter 2, *Application Examples*, contains nine sample applications designed to illustrate specific GPIB concepts and techniques that can help you write your own applications. The description of each example includes the programmer's task, a program flowchart, and numbered steps that correspond to the numbered blocks on the flowchart.

- Chapter 3, *Developing Your Application,* explains how to develop a GPIB application program using NI-488 functions and NI-488.2 routines.

- Chapter 4, *Debugging Your Application,* describes several ways to debug your application program.

- Chapter 5, *ibic—Interface Bus Interactive Control Utility*, introduces you to `ibic`, the interactive control program that you can use to communicate with GPIB devices through functions you enter at your keyboard.

- Chapter 6, *GPIB Programming Techniques*, discusses the following GPIB topics: data transfer termination methods, waiting for GPIB conditions, device-level calls and bus management, serial polling and SRQ servicing, and parallel polling.

- Chapter 7, *ibconf—Interface Bus Configuration Utility*, contains a description of `ibconf`, the software configuration program you can use to configure the NI-488.2M software.

• Appendix A, *Status Word Conditions*, gives a detailed description of the conditions reported in the status word, `ibsta`.

• Appendix B, *Error Codes and Solutions*, lists a description of each error, some conditions under which it might occur, and possible solutions.

• Appendix C, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.

• The *Glossary* contains an alphabetical list and a description of terms, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols, that this manual uses.

• The *Index* contains an alphabetical list of key terms and topics in this manual and it includes the page where you can find each term and topic.

Conventions Used in This Manual

The following conventions are used in this manual.

italic	Italic text denotes emphasis, cross references, field names, or an introduction to a key concept.
bold italic	Bold italic text denotes a note, caution, or warning.
`monospace`	Text in this font denotes text or characters that you enter from the keyboard. Sections of code, programming examples, and syntax examples also appear in this font. This font is also used for the proper name of disk drives, paths, directories, device names, variables, and for statements taken from program code.
`bold monospace`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen.

`italic monospace`	Italic lowercase text in this font denotes that you must supply the appropriate words or values in the place of these items.
<>	Angle brackets enclose the name of a key on the keyboard—for example, <PageDown>.
<Enter>	Key names are capitalized.
-	A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <Control-C>.
enter	*Enter* is reserved to mean that the commands immediately succeeding the word must be typed into the computer and then executed by pressing the <Return> key on the keyboard.
IEEE 488 and IEEE 488.2	IEEE 488 and IEEE 488.2 refer to the ANSI/IEEE Standard 488.1-1987 and the ANSI/IEEE Standard 488.2-1987, respectively, which define the GPIB.
NI-488.2M software	NI-488.2M software refers to the NI-488.2M software for OS/2 unless otherwise noted.

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

How to Use This Manual Set
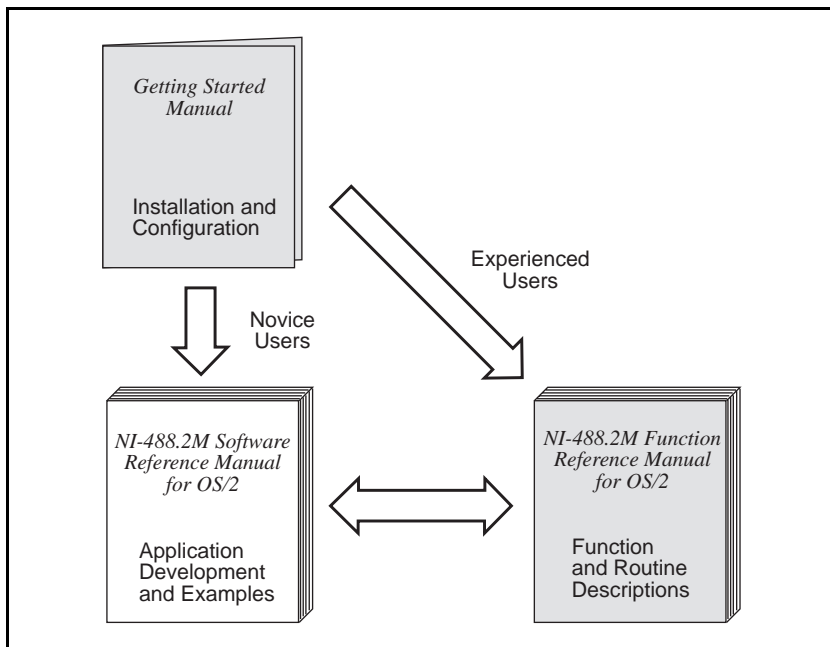
Use the getting-started manual to install and configure your GPIB hardware and NI-488.2M software for OS/2.

Use the software reference manual if you want to learn the basics of GPIB and how to develop an application program. The software reference manual also contains debugging information and detailed examples.

Use the function reference manual for specific information about each NI-488 function and NI-488.2 routine, such as format, parameters, and possible errors.

Related Documentation

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

• *OS/2 Technical Library, Application Design Guide*

• *OS/2 Technical Library, Control Program Programming Reference*

• *OS/2 Technical Library, Programming Guide Volume I*

• *OS/2 Technical Library, Programming Guide Volume II*

• *OS/2 Technical Library, Programming Guide Volume III*

• ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation.*

• ANSI/IEEE Standard 488.2-1987, *IEEE Standard Codes, Formats, Protocols, and Common Commands.*

Customer Communication

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

Chapter 1
NI-488.2M Software Description

This chapter describes the NI-488.2M software package and explains how the software works with your OS/2 system.

The NI-488.2M Software Package

The following section highlights important elements of the NI-488.2M software for OS/2 and describes the function of each element.

NI-488.2M Driver and Driver Utilities

The distribution disk contains the following driver and utility files.

- `readme.doc` is a documentation file that contains important information about the NI-488.2M software and a description of any new features. Before you use the software, read this file for the most recent information.

- `install.cmd` is an OS/2 command file that performs the software installation. It does not modify your `config.sys` file.

- `gpib.sys` is the software driver file that is loaded at system startup by OS/2.

- `gpib.ddp` is a device driver profile. This file is used by the OS/2 command, `ddinstal`, to control the installation process.

- `ibic.exe` is an interactive control program that you use to communicate with the GPIB devices interactively using NI-488.2 functions and routines. It helps you to learn the NI-488.2 routines and to program your instrument or other GPIB devices.

- `ibconf.exe` is a software configuration program that changes the configuration parameters of the NI-488.2M software.

- `ibtest.exe` is the software installation test.

Language Files

The distribution disk contains the following language-related files in the
C subdirectory.

- `readme.doc` is a documentation file that contains information about the
  language interfaces.

- `ni488.dll` is a 32-bit IBM C language interface Dynamic Link Library (DLL)
  file.

- `nibor.dll` is a 32-bit Borland C language interface DLL file.

- `ni488_16.dll` is a 16-bit Microsoft C language interface DLL file.

- `ni488.lib` is an import library for the 32-bit IBM C language interface that you
  must link with your IBM C applications.

- `nibor.lib` is an import library for the 32-bit Borland C language interface that
  you must link with your Borland C applications.

- `ni488_16.lib` is an import library for the 16-bit Microsoft C language
  interface that you must link with your Microsoft C applications.

- `decl.h` is a 32-bit include file. It contains NI-488 functions and NI-488.2
  routine prototypes and various predefined constants.

- `decl_16.h` is a 16-bit include file. It contains NI-488 functions and
  NI-488.2 routine prototypes and various predefined constants.

Example Program Files

The distribution disk contains the following example program files.

- `simple.c` is a C program that illustrates basic communication between a
  computer and a GPIB device.

- `clr_trg.c` is a C program that illustrates how to clear or trigger GPIB devices.

- `asynch.c` is a C program that illustrates how to perform asynchronous I/O.

- `rqs.c` is a C program that illustrates device requests using NI-488 functions.

- `easy4882.c` is a C program that illustrates basic communication with IEEE 488.2 compliant devices using NI-488.2 routines.

- `eos.c` is a C program that shows the use of the end-of-string character.

- `rqs4882.c` is a C program that illustrates serial polls using NI-488.2 routines.

- `ppoll.c` is a C program that illustrates parallel polls using NI-488.2 routines.

- `non_cic.c` is a C program that illustrates communication when the GPIB board is not the Controller.

API-Related Files

The distribution disk contains the following API-related files in the `API` subdirectory.

- `readme.api` is a documentation file that contains information about the OS/2 API functions.

- `nicode.h` is a C language declaration file that contains definitions of NI-488.2M function codes and other NI-488.2M-related constant and structure definitions.

- `nictl_32.h` is a C language declaration file that contains a macro definition. You can use this definition in place of the `DosDevIOCtl` definition for applications that use 32-bit compilers.

- `nictl_16.h` is a C language declaration file that contains a macro definition. You can use this definition in place of the `DosDevIOCtl` definition for applications that use 16-bit compilers.

- `dsamp_32.c` is a 32-bit C sample program using API device-level calls.

- `dsamp_16.c` is a 16-bit C sample program using API device-level calls.

- `bsamp_32.c` is a 32-bit C sample program using API board-level calls.

- `bsamp_16.c` is a 16-bit C sample program using API board-level calls.

How the NI-488.2M Software Works with OS/2

The NI-488.2M driver operates as a standard OS/2 system device driver, which is loaded at system startup. OS/2 device driver services are available to all OS/2 programs running in the system.

Figure 1-1 shows two ways you can access the NI-488.2M driver. You can use either the NI-488.2 language interface or the OS/2 API (Application Program Interface) calls. Both methods access the NI-488.2M driver through the OS/2 system. The driver then accesses your GPIB hardware.
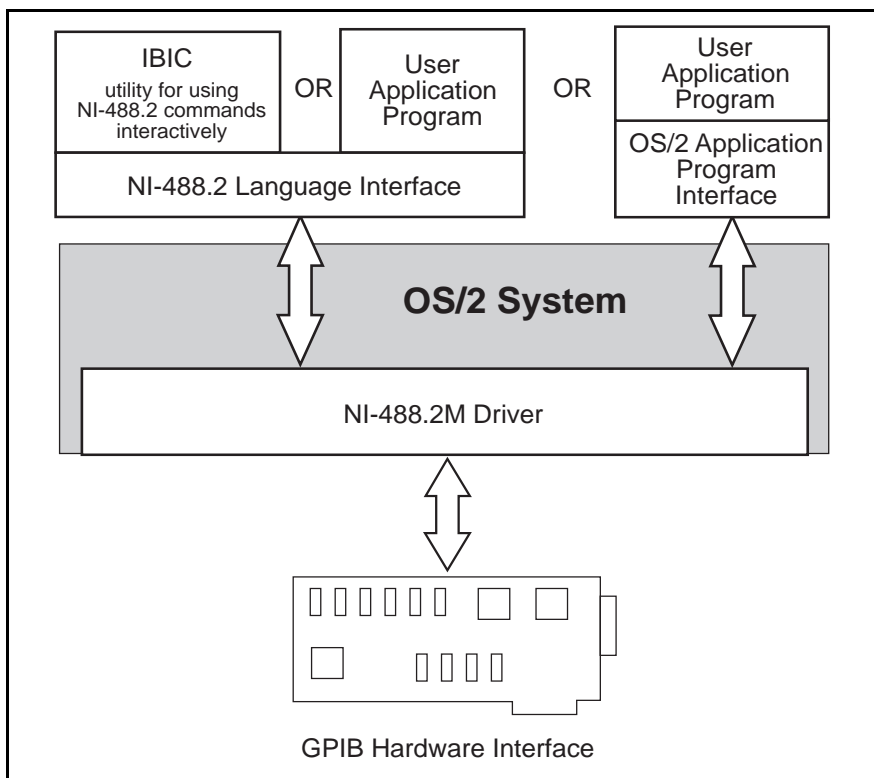
Figure 1-1. How the NI-488.2M Software Works with OS/2

The remainder of this chapter discusses the basics of GPIB and how to set up your system. For application program examples, refer to Chapter 2, *Application Examples*. For information about writing an application program, refer to Chapter 3, *Developing Your Application*.

GPIB Overview

The following sections describe the elements of a GPIB system.

The IEEE 488 Standard and GPIB

The ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation,* describes a standard interface for communication between instruments and controllers from various vendors. It contains information about electrical, mechanical, and functional specifications. The ANSI/IEEE Standard 488.2-1987, *Codes, Formats, Protocols, and Common Commands*, defines a bus communication protocol, a common set of data codes and formats, and a generic set of common device commands.

The GPIB (General Purpose Interface Bus) is a digital, 8-bit parallel communications interface with data transfer rates of 1 Mbytes/s and above. The bus supports one System Controller, usually a computer, and up to 14 additional instruments.

Talkers, Listeners, and Controllers

Devices on the GPIB 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 GPIB interface board and GPIB software installed, it can function as a Talker, Listener, and Controller.

Controller-In-Charge and System Controller

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

Sending Messages Across the GPIB

Devices on the bus communicate by sending messages. Signals and lines transfer these messages across the GPIB interface, which consists of 16 signal lines and 8 ground return (shield drain) lines. The 16 signal lines are discussed in the following sections.

Data Lines

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

Handshake Lines

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

Table 1-1. GPIB Handshake Lines

Line	Description
NRFD (not ready for data)	Listening device is ready/not ready to receive a message byte.
NDAC (not data accepted)	Listening device has/has not accepted a message byte.
DAV (data valid)	Talking device indicates signals on data lines are stable (valid) data.

Interface Management Lines

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

Table 1-2. GPIB Interface Management Lines

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

Setting Up and Configuring Your System

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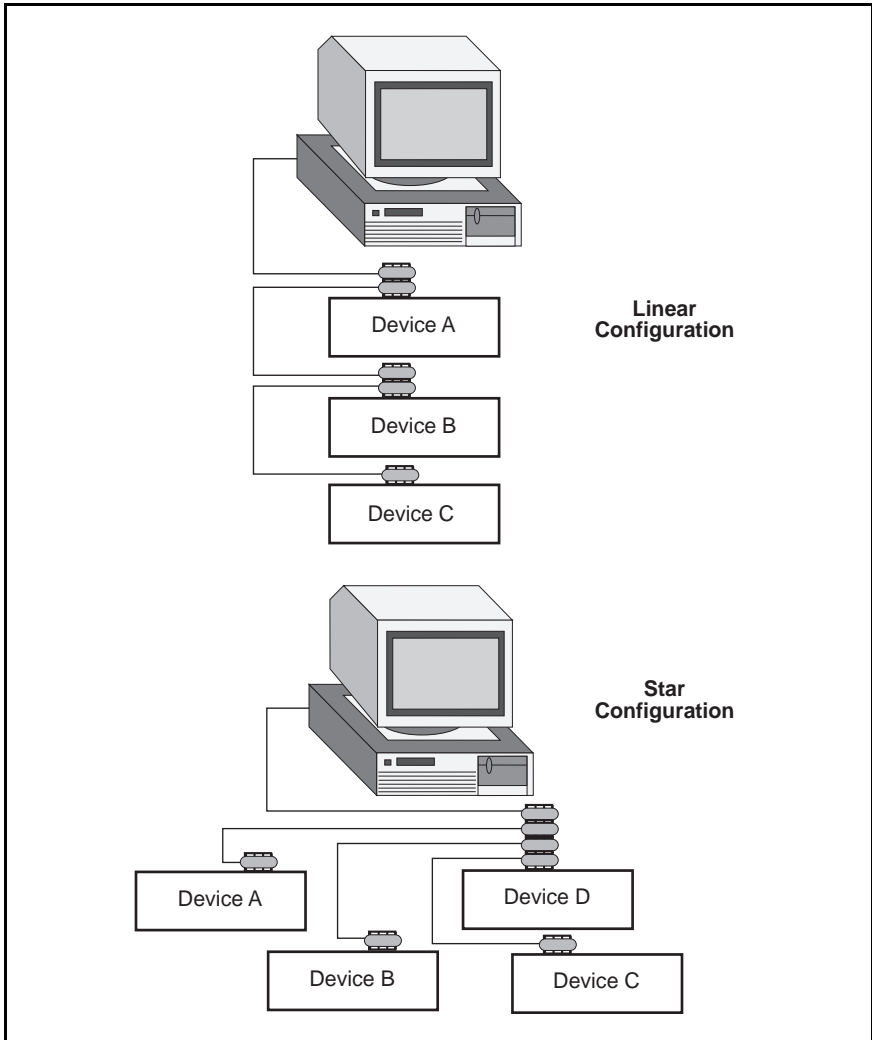
Figure 1-2. Linear and Star System Configuration

Controlling More Than One Board

Multiboard drivers, such as the NI-488.2M driver for OS/2, can control more than one interface board. Figure 1-3 shows an example of a multiboard system configuration.
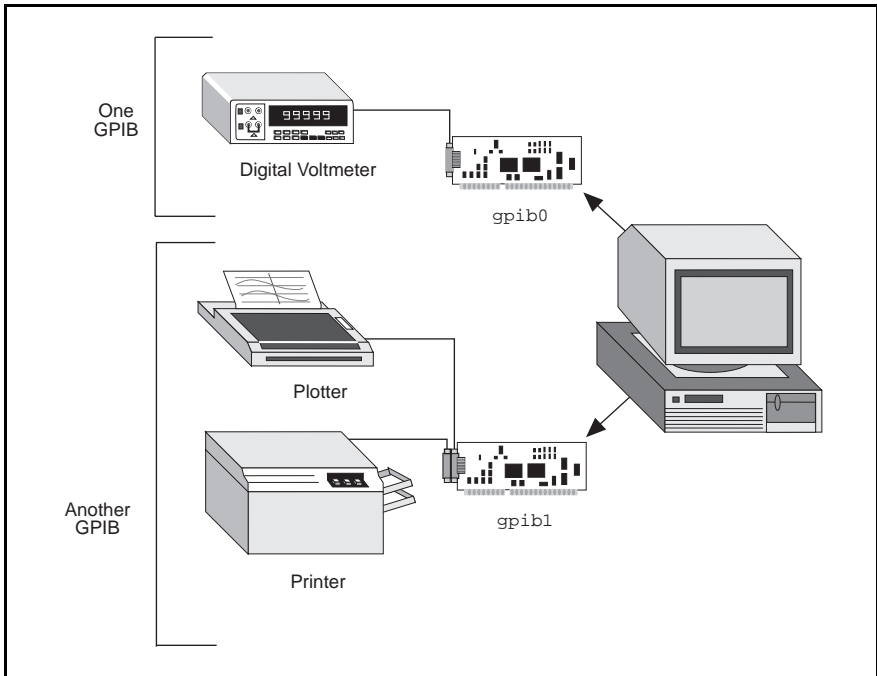
Figure 1-3. Example of Multiboard System Setup

`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.

Chapter 2
Application Examples

This chapter contains nine sample applications designed to illustrate specific GPIB concepts and techniques that can help you write your own applications. The description of each example includes the programmer's task, a program flowchart, and numbered steps that correspond to the numbered blocks on the flowchart.

Use this chapter along with your distribution disk, which contains the C source code for each of the nine examples. If you are new to GPIB programming, you might want to study the contents and concepts of the first sample, `simple.c`, before moving on to more complex examples.

- `simple.c` is the source code file for Example 1. It illustrates how you can establish communication between a host computer and a GPIB device.

- `clr_trg.c` is the source code file for Example 2. It illustrates how you can clear and trigger GPIB devices.

- `asynch.c` is the source code file for Example 3. It illustrates how you can perform non-GPIB tasks while data is being transferred over the GPIB.

- `eos.c` is the source code file for Example 4. It illustrates the concept of the end-of-string (EOS) character.

- `rqs.c` is the source code file for Example 5. It illustrates how you can communicate with GPIB devices that use the GPIB SRQ line to request service. This sample is written by using NI-488 functions.

- `easy4882.c` is the source code file for Example 6. It is an introduction to NI-488.2 routines.

- `rqs4882.c` is the source code file for Example 7. It uses NI-488.2 routines to communicate with GPIB devices that use the GPIB SRQ line to request service.

- `ppoll.c` is the source code file for Example 8. It uses NI-488.2 routines to conduct parallel polls.

- `non_cic.c` is the source code file for Example 9. It illustrates how you can use the NI-488.2M driver in a non-Controller application.

Example 1: Basic Communication

This example focuses on the basics of establishing communication between a host computer and a GPIB device.

A technician needs to monitor voltage readings using a GPIB multimeter. His computer is equipped with an IEEE 488.2 interface board. The NI-488.2M software is installed and a GPIB cable runs from the computer to the GPIB port on the multimeter.

The technician is familiar with the multimeter remote programming command set. This list of commands is specific to his multimeter and is available from the multimeter manufacturer.

He sets up the computer to direct the multimeter to take measurements and record each measurement as it occurs. To do this, he has written an application that uses some simple high-level GPIB commands. The following steps correspond to the program flowchart in Figure 2-1.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application sends the multimeter an instruction, setting it up to take voltage measurements in autorange mode.

3. The application sends the multimeter an instruction to take a voltage measurement.

4. The application tells the multimeter to transmit the data it has acquired to the computer.

    The process of requesting a measurement and reading from the multimeter (Steps 3 and 4) is repeated as long as there are readings to be obtained.

5. As a cleanup step before exiting, the application returns the interface board back to its original state by taking it offline.

Figure 2-1. Program Flowchart for Example 1

Example 2: Clearing and Triggering Devices

This example illustrates how you can clear and trigger GPIB devices.

Two freshman physics lab partners are learning how to use a GPIB digital oscilloscope. They have successfully loaded the NI-488.2M software on a personal computer and connected their GPIB board to a GPIB digital oscilloscope. Their current lab assignment is to write a small application to practice using the oscilloscope and its command set using high-level GPIB commands. The following steps correspond to the program flowchart in Figure 2-2.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application sends a GPIB clear command to the oscilloscope. This command clears the internal registers of the oscilloscope, reinitializing it to default values and settings.

3. The application sends a command to the oscilloscope telling it to read a waveform each time it is triggered. Predefining the task in this way decreases the execution time required. Each trigger of the oscilloscope is now sufficient to get a new run.

4. The application sends a GPIB trigger command to the oscilloscope. The GPIB trigger command causes the oscilloscope to acquire data.

5. The application queries the oscilloscope for the acquired data. The oscilloscope sends the data.

6. The application reads the data from the oscilloscope.

7. The application calls an external graphics routine to display the acquired waveform.

    Steps 4, 5, 6, and 7 are repeated until all the desired data has been acquired by the oscilloscope and received by the computer.

8. As a cleanup step before exiting, the application returns the interface board to its original state by taking it offline.

Figure 2-2. Program Flowchart for Example 2

Example 3: Asynchronous I/O

This example illustrates how an application conducts data transfers with a GPIB device and immediately returns to perform other non-GPIB related tasks while GPIB I/O is occurring in the background. This asynchronous mode of operation is particularly useful when the requested GPIB activity may take some time to complete.

In this example, a research biologist is trying to obtain accurate CAT scans of a lab animal's liver. She will print out a color copy of each scan as it is acquired. The entire operation is computer controlled. The CAT scan machine sends the images it acquires to a computer connected to a GPIB color printer and fitted with the NI-488.2M software package. The biologist is familiar with the command set of her color printer, as described in the user manual provided by the manufacturer. She acquires and prints images with the aid of an application program she wrote using high-level GPIB commands. The following steps correspond to the program flowchart in Figure 2-3.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. An image is scanned in.

3. The application sends the GPIB printer a command to print the new image and immediately returns without waiting for the I/O operation to be completed.

4. The application saves the image obtained to a file.

5. The application inquires as to whether the printing operation has completed by issuing a GPIB wait command. If the status reported by the wait command indicates completion (CMPL is in the status returned) and more scans need to be acquired, Steps 2 through 5 are repeated until the scans have all been acquired. If the status reported by the wait command in Step 5 does not indicate that printing is finished, statistical computations are performed on the scan obtained and Step 5 is repeated.

6. As a cleanup step before exiting, the application returns the interface board back to its original state by taking it offline.

Figure 2-3. Program Flowchart for Example 3

Example 4: End-of-String Mode

This example illustrates how to use the end-of-string modes to detect that the GPIB device has finished sending data.

A journalist is using a GPIB scanner to scan some pictures into his personal computer for a news story. A GPIB cable runs between the scanner and the computer. He is using an application written by an intern in the department who has read the instruction manual provided by the scanner manufacturer and is familiar with its programming requirements. The following steps correspond to the program flowchart in Figure 2-4.

1.   The application initializes the GPIB by bringing the interface board in the computer online.

2.   The application sends a GPIB clear message to the scanner, initializing it to its power-on defaults.

3.   The scanner needs to see a delimiter indicating the end of a command. In this case, the scanner expects the commands to be terminated with <CR><LF> (carriage return, \r, and linefeed, \n). The application sets its end-of-string (EOS) byte to <LF>. The linefeed code indicates to the scanner that there is no more data coming, and is called the end-of-string byte. It flags an end-of-string condition for this particular GPIB scanner. The same effect could be accomplished by asserting the EOI line when the command is sent.

4.   With the exception of the scan resolution, all the default settings are appropriate for the task at hand. The application changes the scan resolution by writing the appropriate command to the scanner.

5.   The scanner sends back information describing the status of the *change resolution* command. This is a string of bytes terminated by the end-of-string character to tell the application it is done changing the resolution.

6.   The application starts the scan by writing the scan command to the scanner.

7.   The application reads the scan data into the computer.

8.   As a cleanup step before exiting, the application returns the interface board back to its original state by taking it offline.
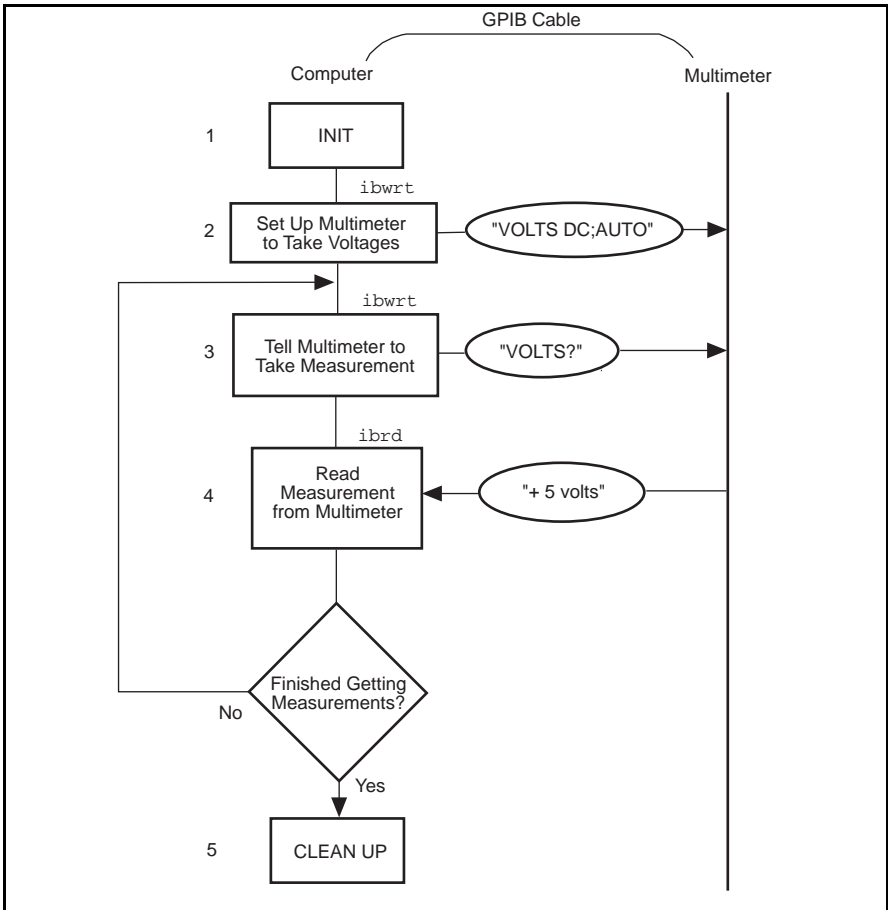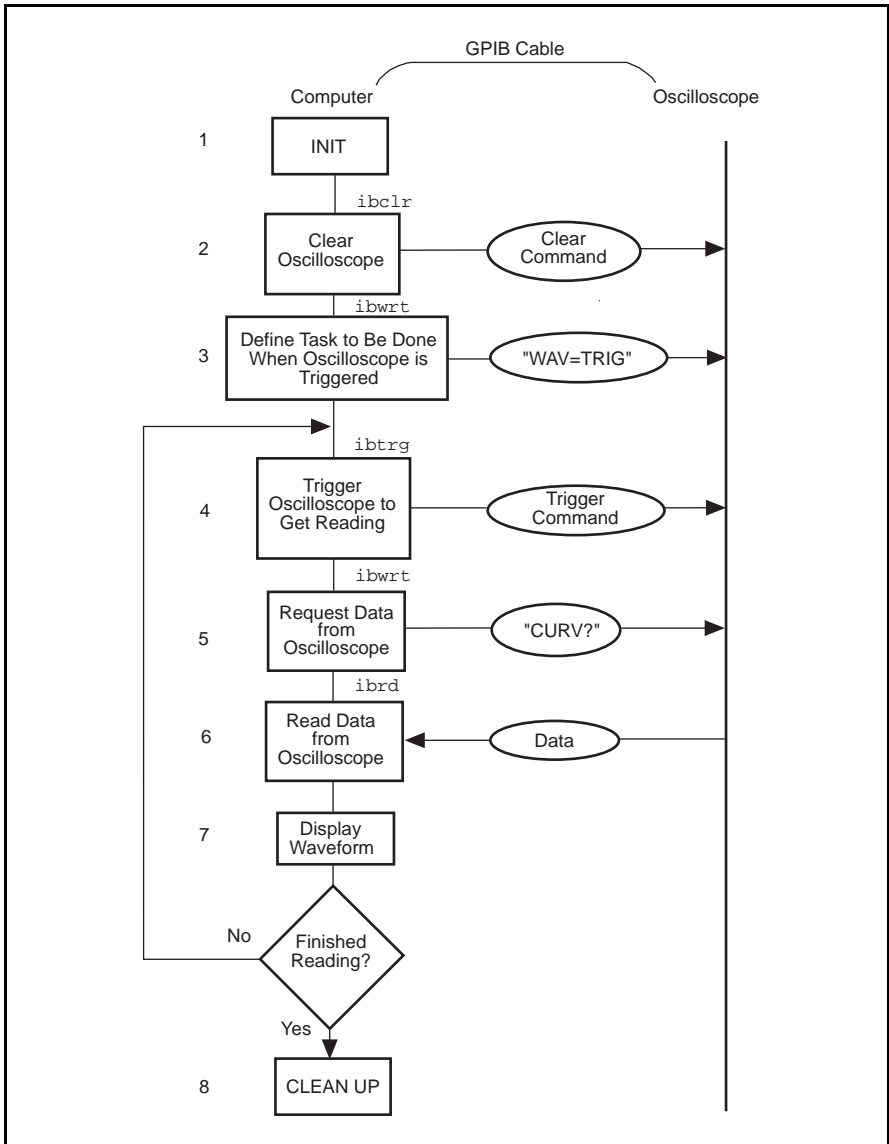
Figure 2-4. Program Flowchart for Example 4

Example 5: Service Requests

This example illustrates how an application communicates with a GPIB device that uses the GPIB service request (SRQ) line to indicate that it needs attention.

A graphic arts designer is transferring digital images stored on her computer to a roll of color film by using a GPIB digital film recorder. A GPIB cable connects the GPIB port on the film recorder to the IEEE 488.2 interface board installed in her computer. She has installed the NI-488.2M software package on the host computer and is familiar with the programming instructions for the film recorder, as described in the user manual provided by the manufacturer. She places a fresh roll of film in the camera and launches a simple application she has written using high-level GPIB commands. With the aid of the application, she records a few images on film. The following steps correspond to the program flowchart in Figure 2-5.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application brings the film recorder to a ready state by issuing a device clear instruction. The film recorder is now set up for operation using its default values. (The graphic arts designer has previously established that the default values for the film recorder are appropriate for the type of film she is using.)

3. The application advances the new roll of film into position so the first image can be exposed on the first frame of film. This is done by sending the appropriate instructions as specified in the film recorder programming guide.

4. The application, by waiting for RQS (request for service), waits for the film recorder to signify that it is done loading the film. The film recorder asserts the GPIB SRQ line when it has finished loading the film.

5. As soon as the film recorder asserts the GPIB SRQ line, the application's wait for the RQS event completes. The application serial polls the device by sending a special command message to the film recorder that directs it to return a response in the form of a serial poll status byte. This byte contains information indicating what kind of service the film recorder is requesting or what condition it is flagging. In this example, it indicates the completion of a command.

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

    a. The application sends a command to the film recorder, directing it to accept data to create a single pass image. The film recorder asserts the SRQ line as soon as a pass is completed.

    b. The application waits for RQS.

    c. When the SRQ line is asserted, the application serial polls the film recorder to see whether it requested service, as in Step 5.

7. The application issues a command to the film recorder to advance the film by one frame. The advance occurs successfully unless the end of film is reached.

8. The application waits for RQS, which completes when the film recorder asserts the SRQ line to signal it is done advancing the film.

9. As soon as the application's wait for RQS completes, the application serial polls the film recorder to see whether it requested service, as in Step 5. The returned serial poll status byte indicates either of two conditions: the film recorder finished advancing the film, as requested, or the end of film was reached and it can no longer advance. Steps 6 through 9 are repeated as long as film is in the camera and more images need to be recorded.

10. As a cleanup step before exiting, the application returns the interface board back to its original state by taking it offline.
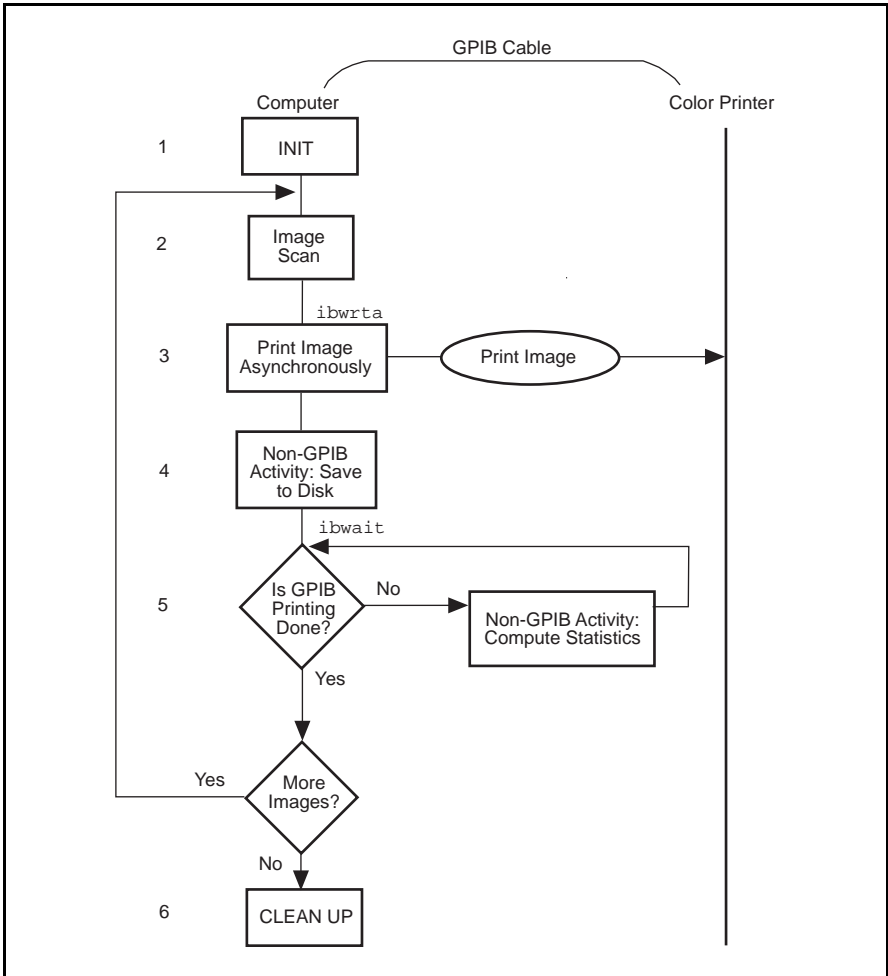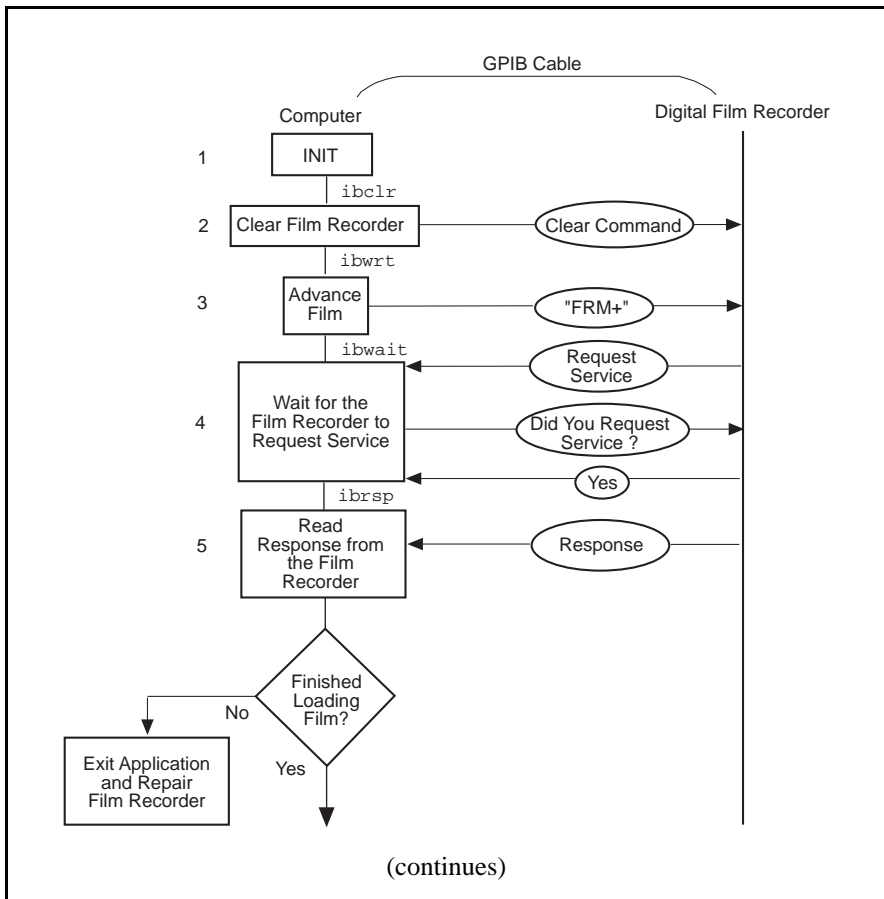
Figure 2-5. Program Flowchart for Example 5
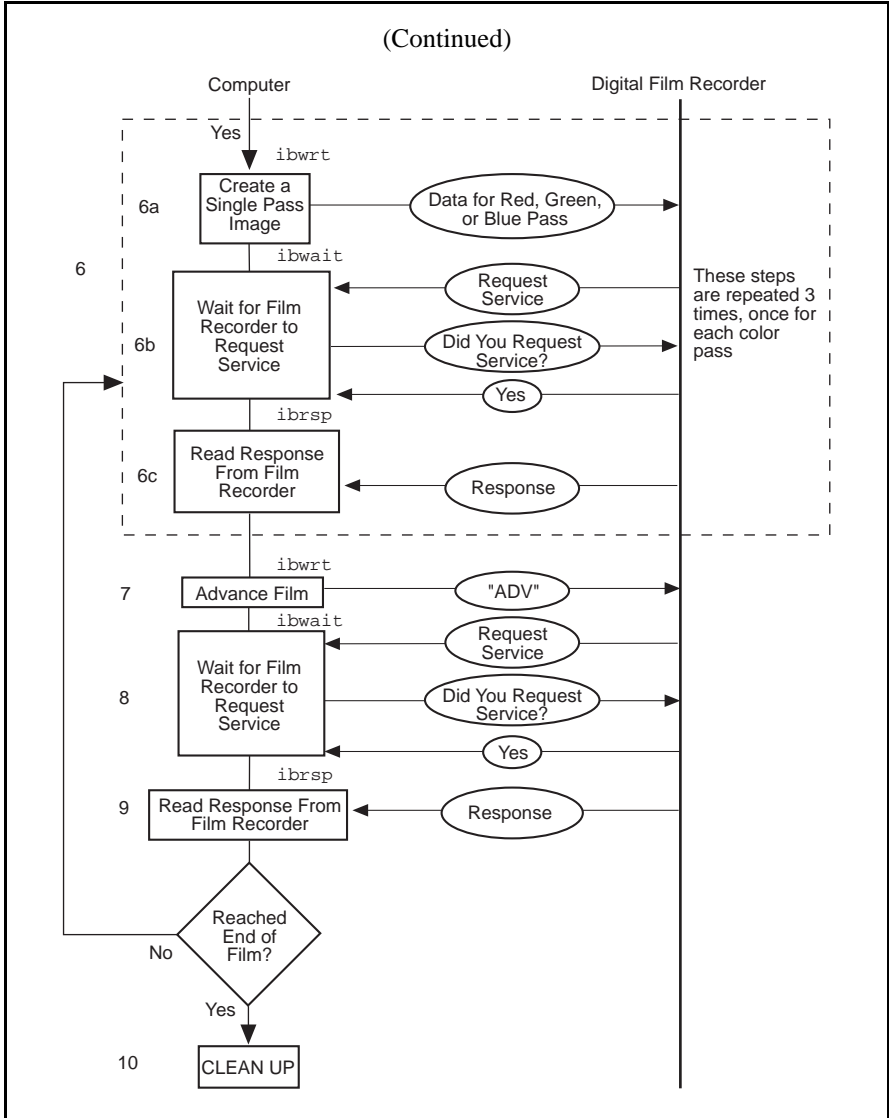
Figure 2-5. Program Flowchart for Example 5 (Continued)

Example 6: Basic Communication with IEEE 488.2 Compliant Devices

This example provides an introduction to communicating with IEEE 488.2 compliant devices.

A test engineer in a metal factory is using IEEE 488.2 compliant tensile testers to find out the strength of metal rods as they come out of production. There are several tensile testers and they are all connected to a central computer equipped with an IEEE 488.2 interface board. These machines are fairly voluminous and it is difficult for the engineer to reach the address switches of each machine. For the purposes of his future work with these tensile testers, he needs to determine what GPIB addresses they have been set to. He can do so with the aid of a simple application he has written. The following steps correspond to the program flowchart in Figure 2-6.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application issues a command to detect the presence of listening devices on the GPIB and compiles a list of the addresses of all such devices.

3. The application sends an identification query (`"*IDN?"`) to a device detected on the GPIB in Step 2.

4. The application reads the identification information returned by the device as it responds to the query in Step 3.

    Steps 3 and 4 are repeated for each of the devices detected in Step 2.

5. As a cleanup step before exiting, the application returns the interface board back to its original state by taking it offline.
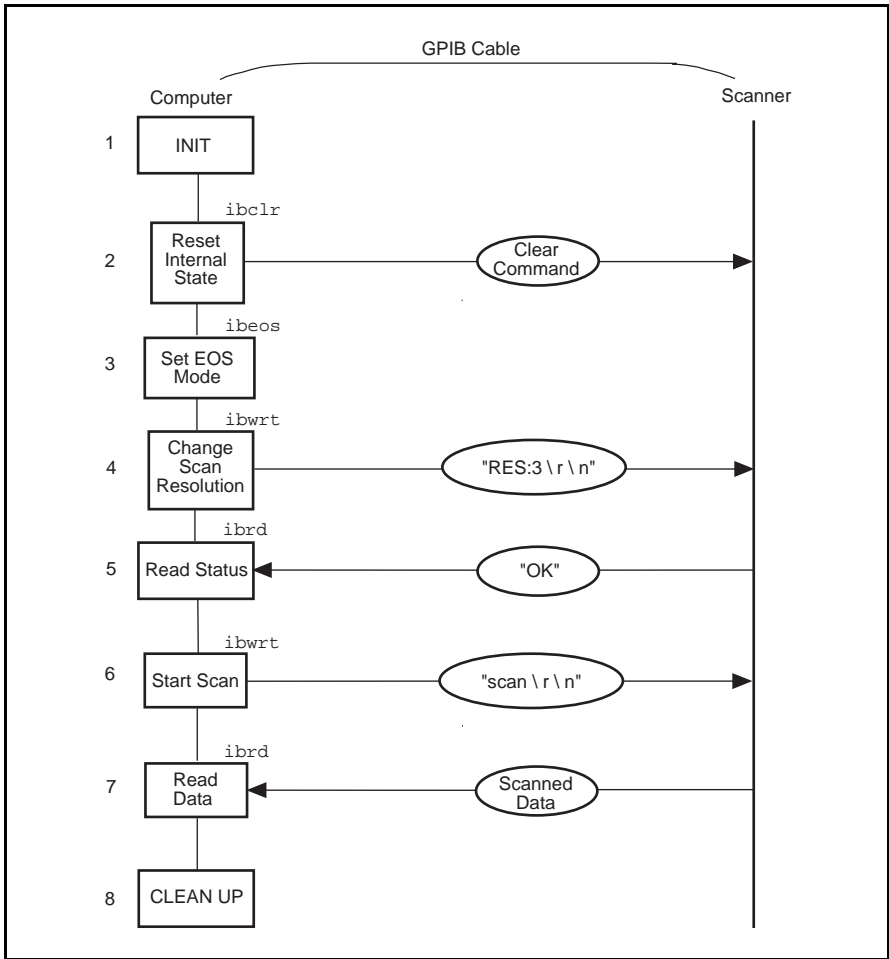
Figure 2-6. Program Flowchart for Example 6

Example 7: Serial Polls Using NI-488.2 Routines

This example illustrates how you can take advantage of the NI-488.2 routines to reduce the complexity of performing serial polls of multiple devices.

A candy manufacturer is using GPIB strain gauges to measure the consistency of the syrup used to make candy. The plant has four big mixers containing syrup. The syrup has to reach a certain consistency to make good quality candy. The consistency is measured by strain gauges that monitor the amount of pressure used to move the mixer arms. When a certain consistency is reached, the mixture is removed and a new batch of syrup is poured in the mixer. The GPIB strain gauges are connected to a computer equipped with an IEEE 488.2 interface board and fitted with the NI-488.2M software. The process is controlled by an application that uses NI-488.2 routines to communicate with the IEEE 488.2 compliant strain gauges. The following steps correspond to the program flowchart in
Figure 2-7.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application configures the strain gauges to request service when they have a significant pressure reading or a mechanical failure occurs. They signal their request for service by asserting the SRQ line.

3. The application waits for one or more of the strain gauges to indicate that they have a significant pressure reading. This wait event ends as soon as the SRQ line is asserted.

4. The application serial polls each strain gauge to see if it requested service.

5. Once the application has determined which strain gauge requires service, it takes a reading from that strain gauge.

6. If the reading matches the desired consistency, a dialog window appears on the computer screen and prompts the mixer operator to remove the mixture and start a new batch. Otherwise, a dialog window prompts the operator to service the mixer in some other way.

    Steps 3 through 6 are repeated as long as the mixers are in operation.

7. After the last batch of syrup has been processed, the application returns the interface board to its original state by taking it offline.
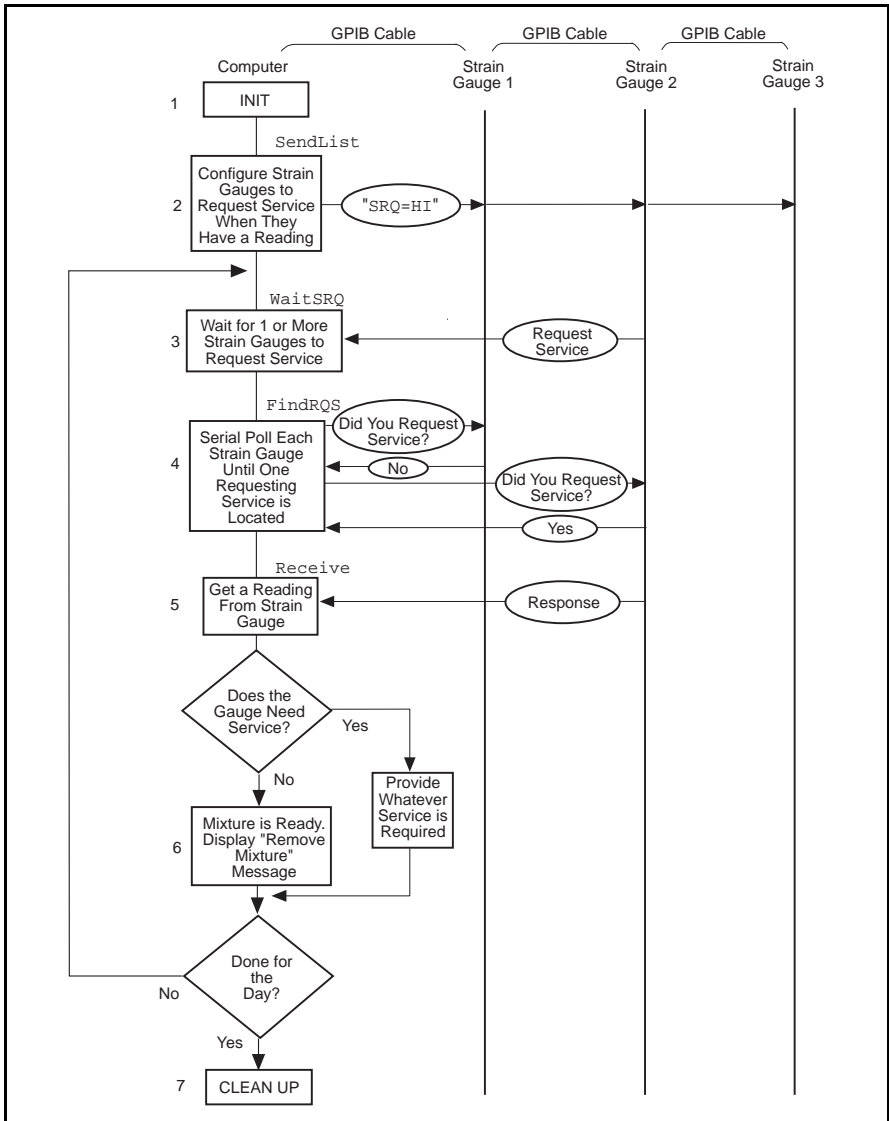
Figure 2-7. Program Flowchart for Example 7

# Example 8: Parallel Polls

This example illustrates how you can use NI-488.2 routines to obtain information from several IEEE 488.2 compliant devices at once by using a procedure called parallel polling.

The process of manufacturing a particular alloy involves bringing three different metals to specific temperatures before mixing them to form the alloy. Three vats are used, each containing a different metal. Each is monitored by a GPIB ore monitoring unit. The monitoring unit consists of a GPIB temperature transducer and a GPIB power supply. The temperature transducer is used to probe the temperature of each metal. The power supply is used to start a motor to pour the metal into the mold when it reaches a predefined temperature. The three monitoring units are connected to the IEEE 488.2 interface board of a computer fitted with the NI-488.2M software and operated by an application using NI-488.2 routines. The application obtains information from the multiple units by conducting a parallel poll, then determines when to pour the metals into the mixture tank. The following steps correspond to the program flowchart in Figure 2-8.

1. The application initializes the GPIB by bringing the interface board in the computer online.

2. The application configures the temperature transducer in the first monitoring unit by choosing which of the eight GPIB data lines the transducer uses to respond when a parallel poll is conducted. The application also sets the temperature threshold. The transducer manufacturer has defined the individual status (`ist`) bit to be true when the temperature threshold is reached, and the configured status mode of the transducer is *assert the data line*. When a parallel poll is conducted, the transducer asserts its data line if the temperature has exceeded the threshold.

3. The application configures the temperature transducer in the second monitoring unit for parallel polls.

4. The application configures the temperature transducer in the third monitoring unit for parallel polls.

5. The application conducts non-GPIB activity while the metals are heated.

6.   The application conducts a parallel poll of all three temperature transducers to determine whether the metals have reached the appropriate temperature. Each transducer asserts its data line during the configuration step if its temperature threshold has been reached.

7.   If the response to the poll indicates that all three metals are at the appropriate temperature, the application sends a command to each of the three power supplies, directing them to power on. Then the motors start and the metals pour into the mold.

     If only one or two of the metals is at the appropriate temperature, Steps 5 and 6 are repeated until the metals can be successfully mixed.

8.   The application unconfigures all the transducers so that they no longer participate in parallel polls.

9.   As a cleanup step before exiting, the application returns the interface board back to its original state by taking it offline.

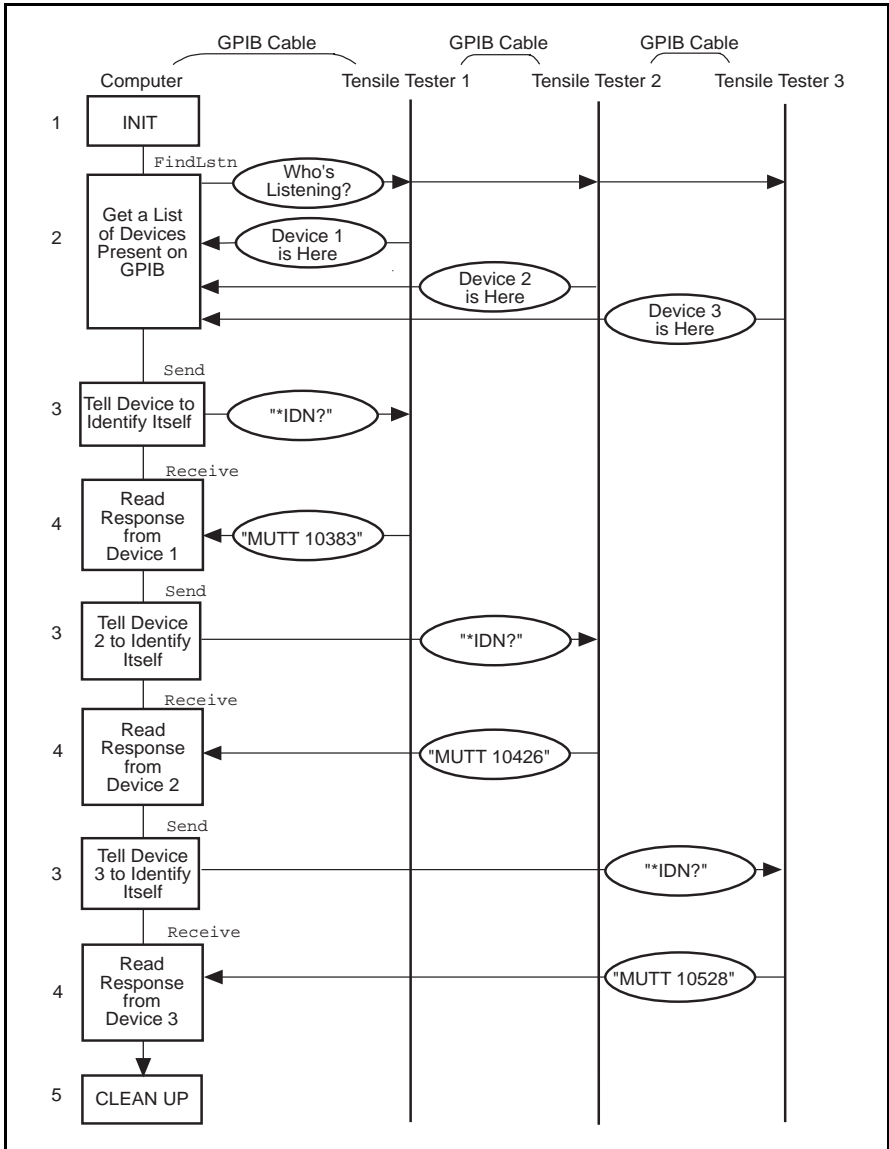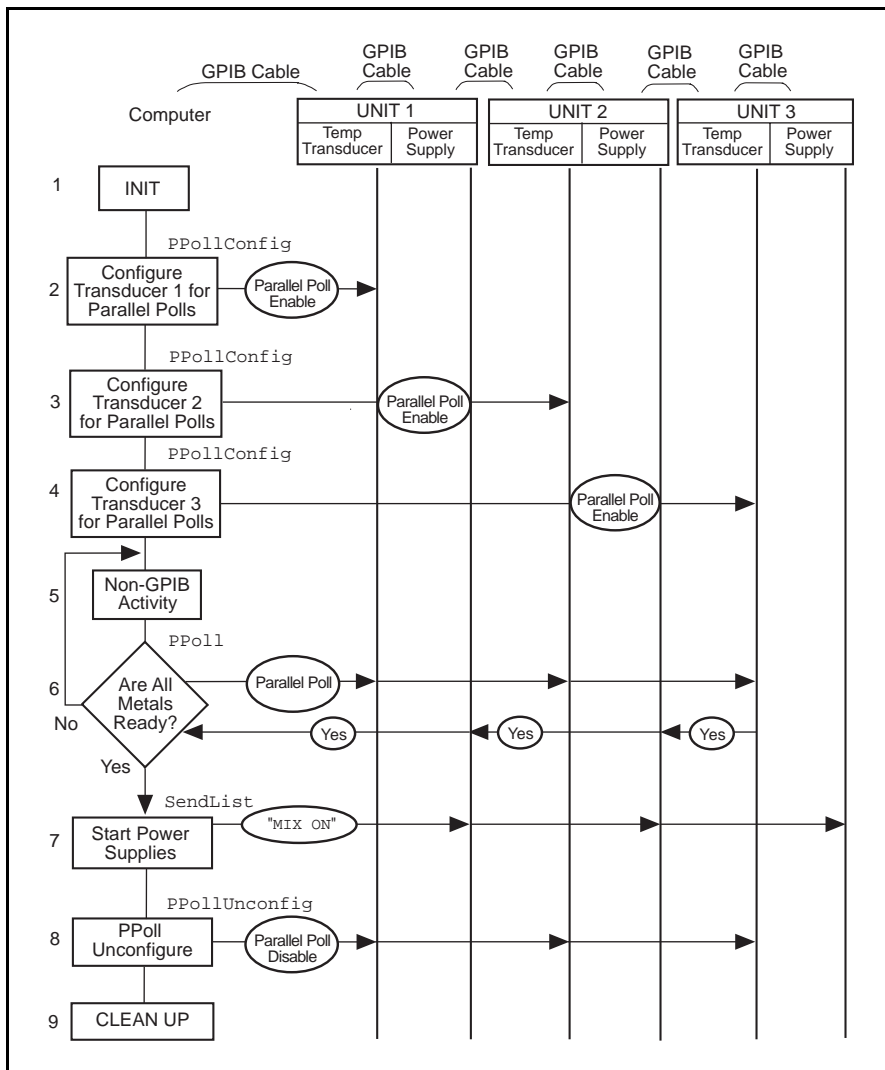Figure 2-8. Program Flowchart for Example 8

Example 9: Non-Controller Example

This example illustrates how you can use the NI-488.2M software to emulate a GPIB device that is not the GPIB Controller.

A software engineer has written firmware to emulate a GPIB device for a research project and is testing it by using an application that makes simple GPIB calls. The following steps correspond to the program flowchart in Figure 2-9.

1. The application brings the device online.

2. The application waits for any of three events to occur: the device becomes listen addressed, becomes talk addressed, or receives a GPIB clear message.

3. As soon as one of the events occurs, the application takes an action based upon the event that occurred. If the device was cleared, the application resets the internal state of the device to default values. If the device is talk addressed, it writes data back to the Controller. If the device is listen addressed, it reads in new data from the Controller.
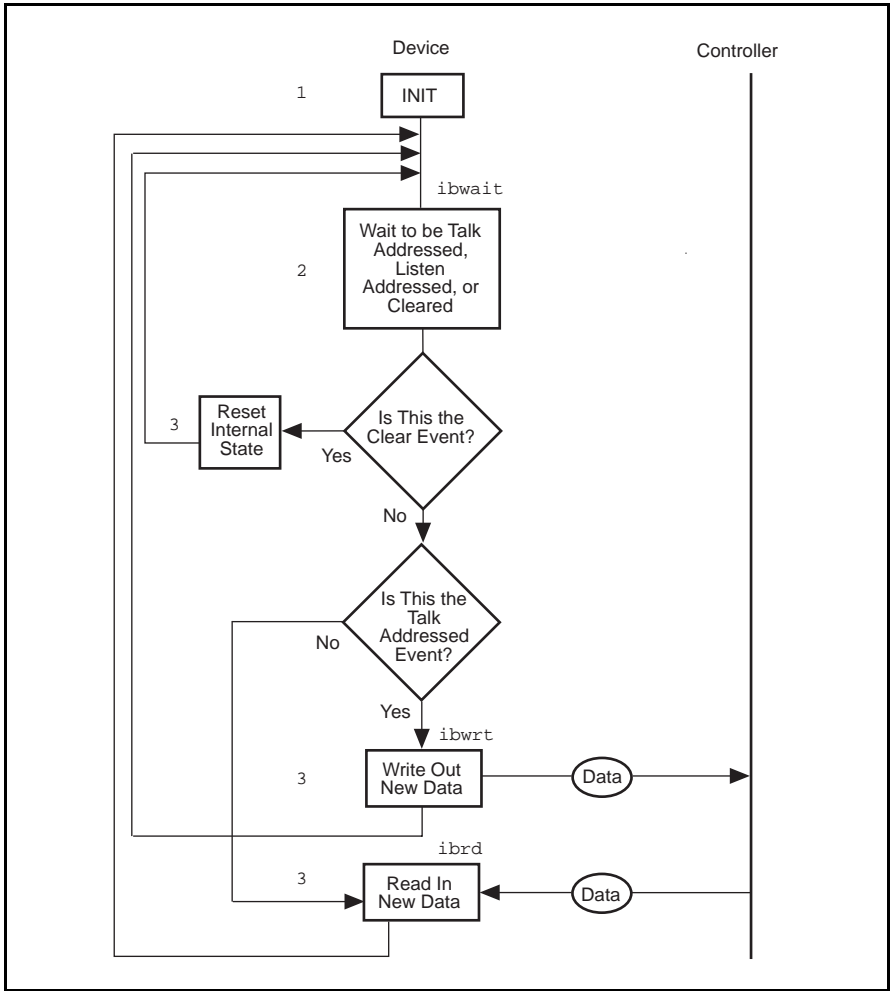
Figure 2-9. Program Flowchart for Example 9

Chapter 3
Developing Your Application

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

Choosing a Programming Method

Programs that need to communicate across the GPIB can access the NI-488.2M driver using either the NI-488.2 language interface or the OS/2 API calls.

Using the NI-488.2 Language Interface

One method of programming the NI-488.2M driver is with an NI-488.2 language interface using functions defined by National Instruments. The NI-488 functions and NI-488.2 routines are an industry standard and are portable across many computer platforms and operating systems. In most cases, you should use these functions and routines because they are designed to make GPIB programming easier. You also should use them if you are already using other National Instruments GPIB products, because the same format and syntax work regardless of the GPIB hardware product. You can make NI-488 or NI-488.2 calls in the `ibic` interactive program or from your application program.

When using the NI-488.2 interface, your OS/2 application runs with both the AT-GPIB driver for OS/2 and MC-GPIB driver for OS/2 without modification or recompiling. Also, multiple applications can share the same library. The NI-488.2M software for OS/2 includes language interface libraries for IBM CSet, Borland C/C++ for OS/2, and Microsoft C 6.0. If you are not programming with one of these languages, you should use the OS/2 API interface.

Your distribution disk contains two distinct sets of subroutines to meet your application needs. Both of these sets, the NI-488 functions and the NI-488.2 routines, are compatible across computer platforms and operating systems, so you can port programs to other platforms with little or no source code

modification. For most application programs, the NI-488 functions are sufficient. You should use the NI-488.2 routines if you have a complex configuration with one or more interface boards and multiple devices. Regardless of which option you choose, the driver automatically addresses and performs other bus management operations necessary for device communication.

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

Using NI-488 Functions: One Device for Each Board

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

*   With NI-488 asynchronous I/O functions (`ibcmda`, `ibrda`, and `ibwrta`), you can initiate an I/O sequence while maintaining control over the CPU for non-GPIB tasks.

*   NI-488 functions include built-in file transfer functions (`ibrdf` and `ibwrtf`).

*   The NI-488 function `ibconfig` dynamically changes the GPIB driver configuration without the need to run the `ibconf` utility.

*   With NI-488 functions, you can control the bus in non-typical ways or communicate with noncompliant devices.

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

NI-488 Device-Level Functions

Device-level functions are high-level functions that automatically execute commands that handle bus management operations such as reading from and writing to devices or polling them for status. If you use device-level functions, you do not need to understand GPIB protocol or bus management. For information about device-level calls and how they manage the GPIB, refer to *Device-Level Calls and Bus Management*, in Chapter 6, *GPIB Programming Techniques.*

NI-488 Board-Level Functions

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

•   Communicating with noncompliant (non-IEEE 488.2) devices

•   Altering various low-level board configurations

•   Managing the bus in non-typical ways

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

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

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

•   Find all of the Listeners on the bus

•   Configure the attached instruments

•   Find a device requesting service

•   Determine the state of the SRQ line

•   Wait for SRQ to be asserted

•   Address multiple devices

• Specify board index, GPIB address, and termination parameters so that you do not need to remember device names, unit descriptors, or termination modes separately

• Use routine names that are descriptive of their purpose

Using the OS/2 API Interface

If the NI-488.2 interface does not meet your requirements, you can access the NI-488.2M driver through the OS/2 API interface. This interface uses the OS/2 standard device driver interface instead of a particular language interface. Because this interface is supported by all devices, you can use it with all development environments. Using the API interface, however, is not as easy as using the NI-488.2 interface. Refer to Chapter 3, *Application Program Interface Function*, in the *NI-488.2M Function Reference Manual for OS/2*, for more information about the API functions.

Checking Status with Global Variables

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

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 NI-488 functions except `ibfind` and `ibdev`. You can test for the conditions reported in `ibsta` and use that information to make decisions about continued processing. Also, if you check for possible errors after each call, debugging your application is much easier.

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

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

Table 3-1. Status Word (ibsta) Layout

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

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

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

or

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

where `gpiberr()` is an error-handling routine that you have defined.

Error Variable—iberr

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

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

For more information on error codes and solutions, refer to Chapter 4, *Debugging Your Application*, or Appendix B, *Error Codes and Solutions.*

Count Variables—ibcnt and ibcntl

The count variables are updated after each read, write, or command function. `ibcnt` is a 16-bit integer and `ibcntl` is a 32-bit integer. If you are reading data, the count variables indicate the number of bytes read. If you are sending data or commands, the count variables reflect the number of bytes sent.

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

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

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

Using ibic to Communicate with Devices

Before you begin writing your application program, you might want to use the
`ibic` utility. With `ibic` (Interface Bus Interactive Control), you communicate
with your instruments from the keyboard rather than from an application
program. You can use `ibic` to learn to communicate with your instruments
using the NI-488 functions or NI-488.2 routines. For specific device
communication instructions, refer to the user manual that came with your
instrument. For information about using `ibic` and for detailed examples, refer
to Chapter 5, *ibic—Interface Bus Interactive Control Utility*.

After you have learned how to communicate with your devices in `ibic`, you are
ready to begin writing your application program.

Writing Your NI-488 Application

This section discusses items you should include in your application program,
general program steps, and examples.

Items to Include

- Include the GPIB header file. This file contains prototypes for the
  NI-488 functions and constants that you can use in your application
  program. Include the declaration file appropriate for your compiler as
  follows:

  ```
  #include "decl.h"          /* 32-bit C compiler */

  #include "decl_16.h"   /* 16-bit C compiler */
  ```

- Check for errors after each NI-488 function call.

- Declare and define a function to handle GPIB errors. This function takes the
  device offline and closes the application. If the function is declared as
  follows

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

  your application invokes the function as follows

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

NI-488 Program Shell

Figure 3-1 is a flowchart of the steps to create your application program using NI-488 functions. The flowchart is for device-level calls.
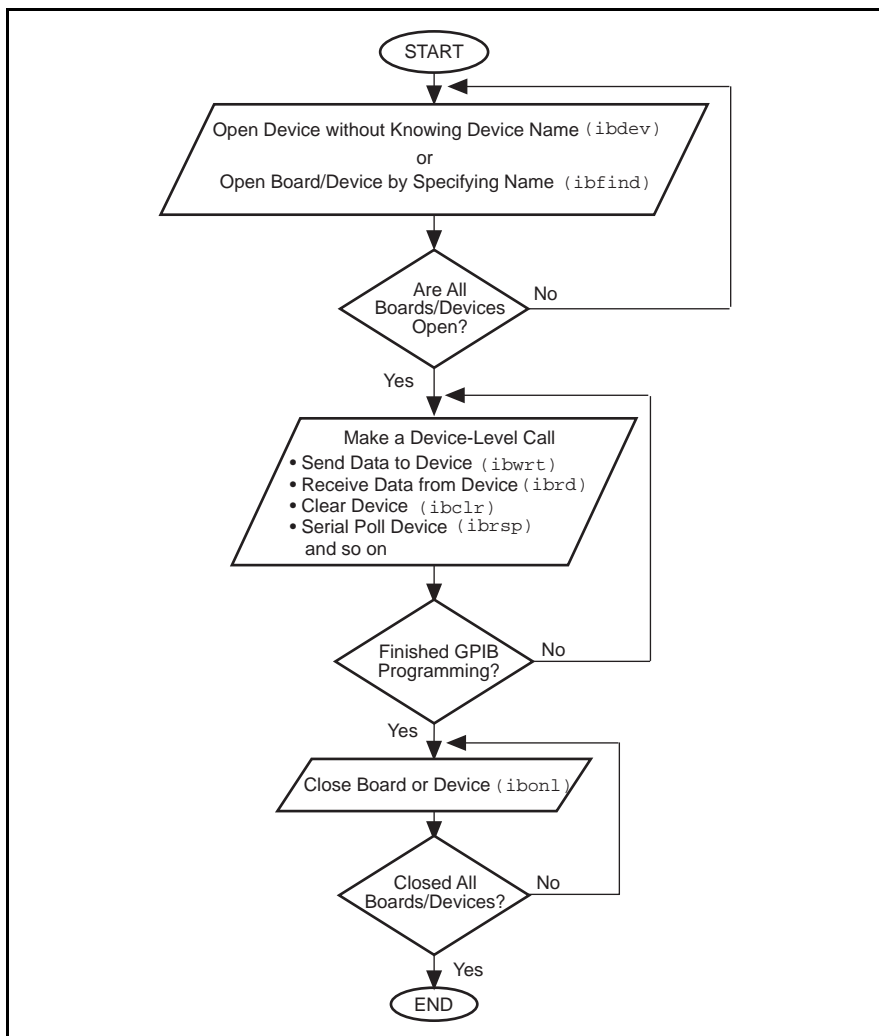
Figure 3-1. General Program Shell Using NI-488 Device Functions

General Program Steps and Examples

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

Step 1. Open a Device

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

```
ud = ibdev(0, 1, 0 , 12, 1, 0);

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

The input arguments of the ibdev function are as follows:

0   –   Board index for gpib0
1   –   Primary GPIB address of the device
0   –   No secondary GPIB address for the device
12 –   I/O timeout value (3 s)
1   –   Send END message with the last byte when writing to the device
0   –   Disable EOS detection mode

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

Step 2. Clear the Device

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

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

Step 3. Configure the Device

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

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

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

Step 4. Trigger the Device

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

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

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

Step 5. Wait for the Measurement

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

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

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

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

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

Step 6. Read the Measurement

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

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

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

sum += AsciiToFloat(rdbuf);
```

Step 7. Process the Data

Repeat Steps 4 through 6 in a loop until 10 measurements have been read, then
print the average of the readings as shown:

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

Step 8. Place the Device Offline

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

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

Writing Your NI-488.2 Application

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

Items to Include

- Include the GPIB header file. This file contains prototypes for the NI-488.2 routines and constants that you can use in your application program. Include the declaration file appropriate for your compiler as follows:

```
#include "decl.h"      /* 32-bit C compiler */

#include "decl_16.h"/* 16-bit C compiler */
```

- Check for errors after each NI-488.2 routine.

- Declare and define a function to handle GPIB errors. This function takes the board offline and closes the application. If the function is declared as follows

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

your application invokes the function as follows

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

NI-488.2 Program Shell

Figure 3-2 is a flowchart of the steps to create your application program using NI-488.2 routines.
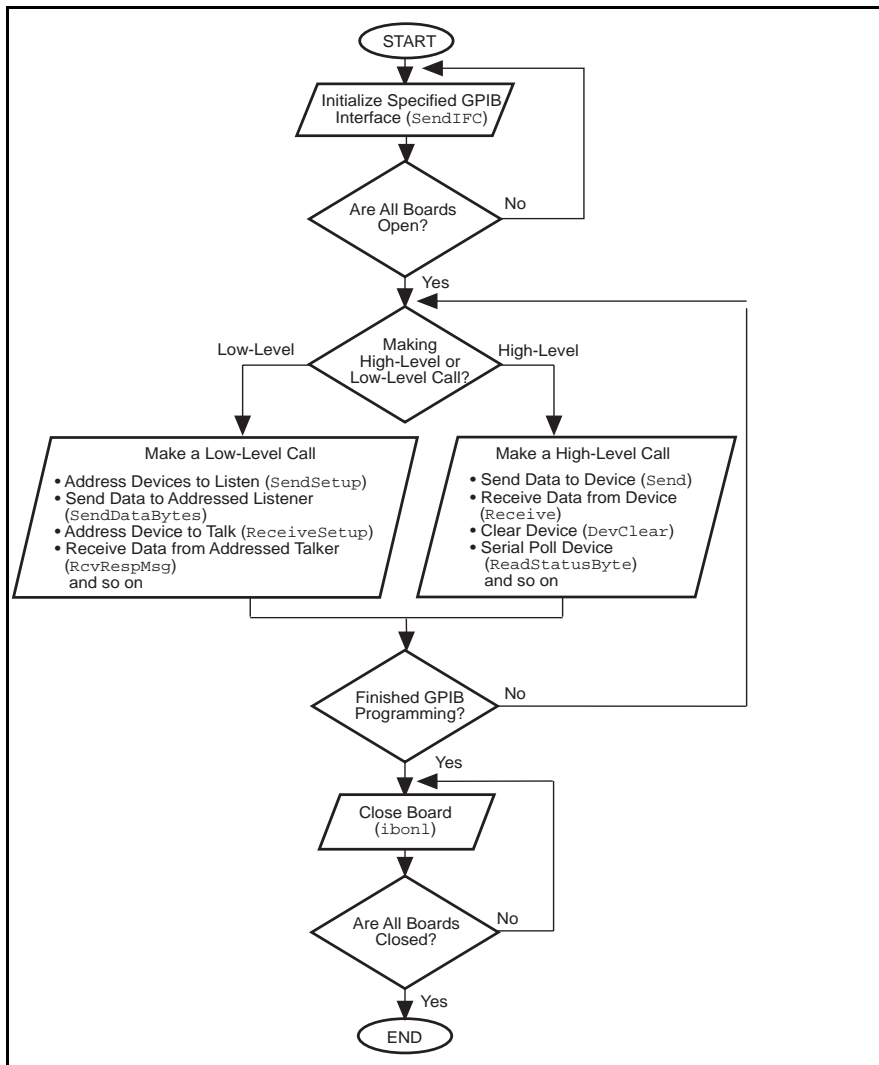
Figure 3-2. General Program Shell Using NI-488.2 Routines

General Program Steps and Examples

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

Step 1. Initialization

Use the `SendIFC` routine to initialize the bus and the GPIB interface board so that the GPIB board is CIC. The only argument of `SendIFC` is the GPIB interface board number.

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

Step 2. Find All Listeners

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

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

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

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

Step 3. Identify the Instrument

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

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

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

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

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

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

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

Step 4. Initialize the Instrument

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

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

Send(0, fluke, "*RST", 4L, NLend);
if  (ibsta & ERR) {
    gpiberr("Send *RST error");
}
```

Step 5. Configure the Instrument

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

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

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

Step 6. Trigger the Instrument

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

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

Step 7. Wait for the Measurement

After the meter is triggered, it takes a measurement and displays it on its front panel, then asserts SRQ. You can detect the assertion of SRQ by using either the `TestSRQ` or `WaitSRQ` routine. If you have a process that you want to execute while you are waiting for the measurement, use `TestSRQ`. For this example, you can use the `WaitSRQ` routine. The first argument in `WaitSRQ` is the GPIB board number. The second argument is a flag returned by `WaitSRQ`. This flag indicates whether SRQ is asserted.

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

After you have detected SRQ, use the `ReadStatusByte` routine to poll the meter and determine its status. The first argument is the GPIB board number, the second argument is the GPIB address of the instrument, and the last argument is a variable that `ReadStatusByte` uses to store the status byte of the instrument.

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

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

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

Step 8. Read the Measurement

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

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

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

Step 9. Process the Data

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

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

Step 10. Place the Board Offline

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

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

Compiling and Linking Your Program

After you have written your application program, you need to compile your
program and link it with the language interface.

Make sure that the full pathname of the import library that you are using
(`ni488.lib`, `nibor.lib`, or `ni488_16.lib`) is included in the `SET LIB`
configuration command in the `config.sys` file. For example, if the import
library is in a directory `d:\at-gpib\c`, the `SET LIB` configuration
command may appear as follows:

```
SET LIB = d:\toolkt20\os2lib;d:\cset2\lib;d:\at-gpib\c
```

32-Bit C Applications

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

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

Compile your 32-bit IBM C program by using the following command:

```
icc  /c  cprog.c
```

then enter the following command to link your compiled program with the NI-
488.2 32-bit IBM C language interface:

```
link386 /NOI cprog.obj,,,ni488.lib;
```

Compile your 32-bit Borland C program using the following command:

```
bcc /c cprog.c
```

then enter the following command to link your compiled program with the NI-488.2 32-bit Borland C language interface:

```
tlink /c /Toe c02.obj cprog.obj,,,os2.lib c2mti.lib
nibor.lib;
```

16-Bit C Applications

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

```
#include "decl_16.h"
```

Compile your 16-bit program by using the following command:

```
cl  /c  cprog.c
```

then enter the following command to link your compiled program with the NI-488.2 16-bit C language interface:

```
link /NOI cprog.obj,,,ni488_16.lib;
```

Running Your Application Program

After you have compiled and linked your application program, you can begin using it. Make sure that the full pathname of the DLL that you are using (`ni488.dll`, `nibor.lib`, or `ni488_16.dll`) is included in the `LIBPATH` statement in your `config.sys` file. For example, if the DLL is in a directory `d:\at-gpib\c`, the `LIBPATH` configuration command may appear as follows:

```
LIBPATH=.;d:\os2\dll;d:\os2\mdos;d:\;d:\at-gpib\c;
```

If you discover errors when you execute the program, refer to Chapter 4, *Debugging Your Application*.

Chapter 4
Debugging Your Application

This chapter describes several ways to debug your application program.

Running ibtest

Before you run your application program, you should run the software
diagnostic test, `ibtest`, that came with your NI-488.2M software. The
`ibtest` program is an NI-488.2M application that makes calls to the driver. If
`ibtest` passes, your GPIB hardware and NI-488.2M software are interacting
correctly. The following paragraphs describe the messages you might receive
while running `ibtest` and how to resolve each problem. The term *GPIBx*
refers to one of the boards `GPIB0`, `GPIB1`, `GPIB2`, and `GPIB3`.

Presence Test of Driver

The following message appears in response to any of four situations:

<<< No driver present for GPIBx. >>>

- The `ibtest` program always tries to test the four AT-GPIB boards. In
  most cases, you have fewer than four boards installed in your computer.
  The above message appears when `ibtest` tries to test a board that does not
  exist. You can ignore this message when it applies to a nonexistent board.

- The GPIB driver might not be installed. To correct this situation, make
  certain that the line `device=y:\at-gpib\gpib.sys`—where `y` refers
  to the letter of the drive where the NI-488.2M software is installed—is in
  your `config.sys` file, then reboot.

- The `Use this GPIB Interface` field in `ibconf` might be set to `no`
  for board `GPIBx`. If you want to use this board, you must set this field to
  `yes`.

• GPIB*x* might be configured to use the same interrupt level that is already used by another device in the system. This situation would cause a conflict that would prevent the driver from installing GPIB*x*. Try a different interrupt level and make certain that the hardware and software are configured to use the same level.

Presence Test of GPIB Board

The following error message appears if GPIB*x* is not installed or if it is not configured at the base I/O address that the driver expects:

<<< No board present for GPIBx. >>>

Check that the board is properly installed in your computer and run ibconf to make certain that the driver is configured to use the same base I/O address as the board.

Incorrect Interrupt Level

The ibtest program outputs dots, then hangs if the AT-GPIB board under test is installed but configured to use an incorrect interrupt level. Run ibconf to configure the driver to use the correct interrupt level.

GPIB Cables Connected

The following error message appears if a GPIB cable was connected to the board when you ran ibtest:

Call(25) 'ibcmd " "' failed, ibsta (0x134) not what was expected (0x8130)

Call(25) 'ibcmd " "' failed, expected ibsta (0x100) to have the ERR bit set.

Disconnect all GPIB cables before trying the test again.

Debugging with the Global Status Variables

After each function call to your NI-488.2M driver, `ibsta`, `iberr`, and `ibcnt` are updated before the call returns to your application. You should check for an error after each GPIB call. Refer to Chapter 3, *Developing Your Application*, for more information about how to use these variables within your program to automatically check for errors.

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

Debugging with ibic

If your application does not automatically check for and display errors, you can locate an error by using `ibic`. Simply issue the same functions or routines, one at a time, as they appear in your application program. Because `ibic` returns the status values and error codes after each call, you should be able to determine which GPIB call is failing. For more information about `ibic`, refer to Chapter 5, *ibic—Interface Bus Interactive Control Utility*.

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

GPIB Error Codes

Table 4-1 lists the GPIB error codes. Remember that the error variable is meaningful only when the ERR bit in the status variable is set. For a detailed description of each error and possible solutions, refer to Appendix B, *Error Codes and Solutions*.

Table 4-1. GPIB Error Codes

Error Mnemonic	iberr Value	Meaning
EDVR	0	OS/2 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
EOIP	10	Asynchronous I/O in progress
ECAP	11	No capability for operation
EFSO	12	File system error
EBUS	14	GPIB bus error
ESTB	15	Serial poll status byte queue overflow
ESRQ	16	SRQ stuck in ON position
ETAB	20	Table problem

Configuration Errors

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

- Application hangs on input or output functions

- Data is corrupted

If these problems occur, make sure that the GPIB hardware settings match the NI-488.2M software settings for the interrupt request level and the DMA channel. For information on hardware and software default settings, refer to the getting-started manual that came with your kit. The next section discusses how to reconfigure your NI-488.2M software, if necessary.

Reconfiguring the NI-488.2M Software

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

`ibconfig` does not permanently change the state of the driver. Using dynamic configuration automatically configures the driver as necessary.

Note: ***To change settings other than base I/O address, interrupt level, or DMA channel, National Instruments recommends using*** `ibconfig` ***instead of running the*** `ibconf` ***utility***.

If your program uses dynamic configuration, it will always work regardless of the previous driver configuration. For more information, refer to the description of `ibconfig` in the *NI-488.2M Function Reference Manual for OS/2*.

Timing Errors

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

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

Communication Errors

Repeat Addressing

Some devices require GPIB addressing before any GPIB activity. Devices adhering to the IEEE 488.2 standard should remain in their current state until specific commands are sent across the GPIB to change their state. You might need to configure your NI-488.2M driver to perform repeat addressing if your device does not remain in its currently addressed state. Refer to Chapter 7, *ibconf—Interface Bus Configuration Utility*, or to the description of `ibconfig` in the *NI-488.2M Function Reference Manual for OS/2* for more information about reconfiguring your software.

Termination Method

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

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

Chapter 5
ibic—Interface Bus Interactive Control Utility

This chapter introduces you to `ibic`, the interactive control program that you can use to communicate with GPIB devices through functions you enter at your keyboard.

Overview

With the Interface Bus Interactive Control (`ibic`) program, you communicate with the GPIB devices through functions you enter at the keyboard. For specific communication instructions, refer to the manual that came with your instrument. You can use `ibic` to practice communication with the instrument, troubleshoot problems, and develop your application program.

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

*   The results of the status word (`ibsta`) in hexadecimal.

*   The mnemonic constant of each bit set in `ibsta`.

*   The mnemonic value of the error variable (`iberr`) if an error exists (the ERR bit is set in `ibsta`).

*   The count value for each read, write, or command function.

*   The data received from your instrument.

Starting ibic

The `ibic` program is contained in `ibic.exe`, the executable file that was copied from your distribution disk when you installed the NI-488.2M software. To run `ibic`, change to the appropriate subdirectory (AT-GPIB in the following example) and enter `ibic` at the prompt as shown:

```
C:/AT-GPIB>       ibic



National Instruments
IEEE-488 Interface Bus Interactive Control Program (IBIC)
Copyright (c) 1993 National Instruments Corp. Version 2.0
Version Date: Apr 30 1993  Version Time: 09:42:25
All Rights Reserved

Press 'help' for help or 'q' to quit.
```

Exiting ibic

Typing e or q terminates execution of the `ibic` program.

ibic Syntax

The syntax of functions in `ibic` differs from the syntax in a programming language. The main difference is that in `ibic`, certain messages (`ibwrt`, `ibwrta`, `ibrd`, `ibrda`, `ibcmd`, `ibcmda`, `Send`, `SendList`, and `SendCmds`) are entered as strings from the keyboard.

Another difference is that the board or device descriptor (`ud`) is not explicitly part of `ibic` function syntax. Before using any device or board, first call `ibfind` to open that unit. `ibic` uses the descriptor returned by the driver for

all subsequent calls to that unit. When the device or board is opened, the symbolic name of that device or board is added to the prompt.

The `ibic` utility makes no distinction between uppercase and lowercase.

Tables 5-1 and 5-2 summarize the syntax of NI-488 functions and NI-488.2 routines that are called from `ibic`. Syntax rules for the functions and routines in `ibic` are explained in the table notes.

Table 5-1. Syntax for NI-488 Functions in ibic

Syntax	Description	Type	Note
`ibask mna`	Return configuration information	dev, brd	16
`ibbna brdname`	Change access board of device	dev	1
`ibcac [v]`	Become active Controller	brd	2,3
`ibclr`	Clear specified device	dev	
`ibcmd string`	Send commands from string	brd	4
`ibcmda string`	Send commands asynch. from string	brd	4
`ibconfig mn v`	Alter configurable parameters	dev, brd	15,3
`ibdev vvvvvv`	Open an unused device when the device name is unknown	dev	9
`ibdma [v]`	Enable/disable DMA	brd	2,3
`ibeos v`	Change/disable EOS message	dev, brd	3
`ibeot [v]`	Enable/disable END message	dev, brd	2,3
`ibfind udname`	Return unit descriptor	dev, brd	5
`ibgts [v]`	Go from active Controller to standby	brd	2,3
`ibist [v]`	Set/clear `ist`	brd	2,3
`iblines`	Read the state of all GPIB lines	dev, brd	
`ibln v v`	Check for presence of device of bus	dev, brd	10
`ibloc`	Go to local	dev, brd	

(continues)

Table 5-1. Syntax for NI-488 Functions in ibic (Continued)

Syntax	Description	Type	Note
`ibonl [v]`	Place device online or offline	dev, brd	2,3
`ibpad v`	Change primary address	dev, brd	3
`ibpct`	Pass control	dev	
`ibppc v`	Parallel poll configure	dev, brd	3
`ibrd v`	Read data	dev, brd	6
`ibrda v`	Read data asynchronously	dev, brd	6
`ibrdf flname`	Read data to file	dev, brd	7
`ibrpp`	Conduct a parallel poll	dev, brd	
`ibrsc [v]`	Request/release system control	brd	2,3
`ibrsp`	Return serial poll byte	dev	
`ibrsv v`	Request service	dev	3
`ibsad v`	Change secondary address	dev, brd	3
`ibsic`	Send interface clear	brd	
`ibsre [v]`	Set/clear remote enable line	brd	2,3
`ibstop`	Abort asynchronous operation	dev, brd	
`ibtmo v`	Change/disable time limit	dev, brd	3
`ibtrg`	Trigger selected device	dev	
`ibwait [mask]`	Wait for selected event	dev, brd	2,8
`ibwrt string`	Write data	brd	4
`ibwrta string`	Write data asynchronously	dev, brd	4
`ibwrtf flname`	Write data from a file	dev, brd	7

Table 5-2. Syntax for NI-488.2 Routines in ibic

Routine Syntax	Description	Note
`AllSpoll list`	Serial poll multiple devices	11
`DevClear address`	Clear a device	13
`DevClearList list`	Clear multiple devices	11
`EnableLocal list`	Enable local control	11
`EnableRemote list`	Enable remote control	11
`FindLstn list v`	Find all Listeners	3,11
`FindRQS list`	Find device asserting SRQ	11
`PassControl address`	Pass control to a device	13
`PPoll`	Parallel poll devices	
`PPollConfig addr.line sense`	Configure device for parallel poll	13,14
`PPollUnconfig address`	Unconfigure device for parallel poll	13
`RcvRespMsg address data mode`	Receive response message	4,12,13
`ReadStatusByte address`	Serial poll a device	13
`Receive address data mode`	Receive data from a device	4,12,13
`ReceiveSetup address`	Receive setup	13
`ResetSys list`	Reset multiple devices	11
`Send address data mode`	Send data to a device	4,12,13
`SendCmds data`	Send command bytes	4
`SendDataBytes list data mode`	Send data bytes	4,11,12
`SendIFC`	Send interface clear	
`SendList list data mode`	Send data to multiple devices	4,11,12
`SendLLO`	Put devices in local lockout	

(continues)

Table 5-2. Syntax for NI-488.2 Routines in ibic (Continued)

Routine Syntax	Description	Note
`SendSetup list`	Send setup	11
`Set 488.2 v`	Enter into 488.2 mode for board	
`SetRWLS list`	Put device in remote with lockout state	11
`TestSys list`	Cause multiple devices to perform self tests	11
`TestSRQ`	Test for service request	
`Trigger address`	Trigger a device	13
`TriggerList list`	Trigger multiple devices	11
`WaitSRQ`	Wait for service request	

Notes for Tables 5-1 and 5-2

1.   `brdname` is the symbolic name of the new board (for example, `gpib1`).

2.   Values enclosed in square brackets (`[ ]`) are optional. The default value is 0 for `ibwait` and 1 for all other functions.

3.   `v` is a hex, octal, or decimal integer. Hex numbers must be preceded by 0 and x (for example, `0xD`). Octal numbers must be preceded by 0 only (for example, `015`). Other numbers are assumed to be decimal.

4.   `string` consists of a list of ASCII characters, octal or hex bytes, or special symbols. You must enclose the entire sequence of characters in quotation marks. An octal byte consists of a backslash character followed by the octal value. For example, octal 40 is represented by `\40`. A hex byte consists of a backslash character and a character `x` followed by the hex value. For example, hex 40 is represented by `\x40`. The two special symbols, `\r` for a carriage return character and `\n` for a linefeed character, are a more convenient method for inserting the carriage return and linefeed characters into the string, as shown in the following example: `"F3R5T1\r\n"`. Because the carriage return is represented equally well in hex, `\xD` and `\r` are equivalent strings.

5.   `udname` is the symbolic name of the new device or board (for example, `dev1` or `gpib0`).

6.   `v` is the number of bytes to read.

7.   `flname` is the path name of the file to be read or written (for example, `\test\meter` or `printr.buf`).

8.   `mask` is a hex, octal, or decimal integer (see note 3), or a mask bit mnemonic.

9.   `ibdev` parameters are `board id`, `pad`, `sad`, `tmo`, `eos`, and `eot`.

10.   `ibln` parameters are `pad` and `sad`.

11.   `list` is a comma-separated list of address integers, optionally enclosed in parentheses. An empty list can be expressed by empty parentheses.

12.   `mode` is a termination mode mnemonic or integer. Mnemonics are `NLend`, `DABend`, and `NULLend` for send operations, and `STOPend` for receive operations.

13.   `address` is an integer representing a GPIB address. If only a primary GPIB address is required, enter that integer. If a secondary address is also required, create an integer with the primary address in the low-order byte, and the secondary address in the high-order byte; for example, `pad` 3 and `sad` 0x61 could be expressed as 0x6103.

14.   `line` and `sense` are integers representing the data line to respond on and the sense of the response.

15.   `mn` is a mnemonic for a configuration parameter or the equivalent integer value. Refer to the description of `ibconfig` in the *NI-488.2M Function Reference Manual for OS/2* for the allowed mnemonics and their values.

16.   `mna` is a mnemonic for a configuration parameter or the equivalent integer value. Refer to the description of `ibask` in the *NI-488.2M Function Reference Manual for OS/2* for the allowed mnemonics and their values.

Adding End-of-String Characters

Some GPIB instruments require special termination characters or end-of-string
(EOS) characters to indicate to the device that a transmission has ended. If your
device requires any EOS characters, you must add these to the end of the data
string sent out by the ibwrt statement. The following example illustrates the
addition of the two most commonly used EOS characters: the carriage return
and the linefeed.

```
dev1:      ibwrt "F3R5T1\r\n"
[0100] (cmpl)
count: 8
```

The \r and \n represent the carriage return and linefeed characters,
respectively. Refer to Appendix A, *Multiline Interface Messages*, in the
NI-488.2M Function Reference Manual for OS/2 for more information about
non-printable characters.

Status Word Return

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

```
dev1:      ibwrt "f2t3x"
[0100] (cmpl)
count: 5

dev1:
```

For more information about the status word, refer to Chapter 3, *Developing Your Application*.

Error Codes Return

If an NI-488 function or NI-488.2 routine completes with an error, `ibic` displays the error mnemonic. The following example illustrates the result if an error condition occurs in a data transfer.

```
dev1:        ibwrt "f2t3x"
[8100] (err cmpl)
error: ENOL
count: 1


dev1:
```

In this example, there are no Listeners, indicating that `dev1` is powered off or the GPIB cable is disconnected. For a detailed list of the error codes and their meanings, refer to Chapter 4, *Debugging Your Application*.

Count Return

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

If one of the addresses in an address list of an NI-488.2 routine is invalid, `ibic` displays the index of the invalid address as the count.

The count return has a different meaning, depending on which NI-488 function or NI-488.2 routine is called. Refer to the function descriptions in the *NI-488.2M Function Reference Manual for OS/2* for the correct interpretation of the count return.

Common NI-488.2 Routines in ibic

Use the auxiliary function `set` to select the NI-488.2 function mode. The syntax for this form of the `set` command is as follows:

```
set 488.2 [n]
```

where n represents an optional board number (for example, n = 1 for `gpib1`). The default value of n is 0 (`gpib0`).

After issuing this form of the `set` command, `ibic` uses the `488.2` prompt to indicate that you are in NI-488.2 mode on board n.

```
        set 488.2 1
        488.2 (1):
```

After issuing the `set 488.2` command, you can use any of the NI-488.2 routines.

Send

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

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

The returned status word [0128] indicates a successful I/O completion, while the byte count indicates that all six characters, including the added new line, were sent from the computer and received by both devices.

Receive

The Receive routine causes the GPIB board to receive data from another GPIB device. The following example illustrates the use of the Receive routine.

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

The command acquires data from the device at primary address 5. It stops receiving data when 10 characters have been received or when the END message is received. The acquired data is then displayed in hex format along with its ASCII equivalent. The ibic program also displays the status word and the count of transferred bytes.

Common NI-488 Functions in ibic

ibfind

To execute any NI-488 function in ibic, you must first use ibfind to open the device or board you want to communicate with. When the device or board is opened, the symbolic name of that device or board is added to the prompt. The unit descriptor of a board or device is returned.

The following example shows ibfind opening dev1.

```
: ibfind dev1
id = 32259

dev1:
```

The returned value is the unit descriptor of the board.

The name used with the ibfind function must be a valid symbolic name known by the driver. dev1 is the default name found in the driver. For more information about valid names, refer to Chapter 7, *ibconf—Interface Bus Configuration Utility*.

ibdev

The ibdev command selects an unopened device, opens it, and initializes its access board.

With ibdev, you can input the values in the following fields:

- Access Board for the Device

- Primary Address

- Secondary Address

- Timeout Setting

- EOT mode

- EOS mode

The following example shows ibdev opening an available device and assigning it to access gpib0 (board = 0) with a primary address of 6 (pad = 6), a secondary address of hex 67 (sad = 0x67), a timeout of 10 ms (tmo = 7), the END message enabled (eot = 1), and the EOS mode disabled (eos = 0).

```
: ibdev 0 6 0x67 7 1 0
id = 32259

ud0:
```

If you type ibdev at the prompt, ibic asks you for the input parameters, as shown in the following example.

```
: ibdev
    enter board index:    0
    enter primary address:    6
    enter secondary address:    0x67
    enter timeout:    7
    enter 'EOI on last byte' flag:    1
    enter end-of-string mode/byte:    0
id = 32259

ud0:
```

Three distinct errors can occur with the ibdev call:

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

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

:
```

- ENEB – The board index entered refers to a known board (such as 0), but the driver cannot find the board. In this case, run ibconf to verify that the base address of the board is set correctly.

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

```
: ibdev 0 66 0x67 7 1 0
id=32261
[8100]  ( err cmpl )
error: EARG


ud0:    ibpad 6
previous value: 16
```

ibwrt

The ibwrt command sends data from one GPIB device to another. For example, to send the six-character data string F3R5T1 from the computer to a device called dev1, you enter the following string at the dev1: prompt.

```
dev1: ibwrt "F3R5T1"
[0100] (cmpl)
count:  6
```

The returned status word [0100] indicates a successful I/O completion, while the byte count indicates that all six characters were sent from the computer and received by the device.

ibrd

The ibrd command causes a GPIB device to receive data from another GPIB device. The following example illustrates the use of the ibrd function.

```
dev1:   ibrd 20
[2100] (end cmpl)
count: 18
4e 44 43 56 28 30 30 30        N D C V ( 0 0 0
2e 30 30 34 37 45 2b 30        . 0 0 4 7 E + 0
0d 0a                          •  •
```

This command acquires data from the device and displays it on the screen in hex format and in its ASCII equivalent, along with information about the data transfer, such as the status word and the byte count.

Auxiliary Functions

Table 5-3 summarizes the auxiliary functions that you can use in `ibic`.

Table 5-3. Auxiliary Functions in ibic

Function	Description	Notes
set udname	Select active device or board	1, 2
help [option]	Display help information	3
!	Repeat previous function	
–	Turn display off	
+	Turn display on	
n* function	Execute function n times	4
n* !	Execute previous function n times	
$ filename	Execute indirect file	5
print string	Display string on screen	6
buffer option	Set the type of display used for buffers	7
e	Exit or quit	
q	Exit or quit	

Notes

1. `udname` is the symbolic name of the new device or board (for example, `dev1` or `gpib0`).

2. Initially call `ibfind` to open each device or board.

3. If `option` is omitted, a menu of options appears.

4. Replace `function` with correct `ibic` function syntax.

5. `filename` is the path name of a file that contains `ibic` functions to be executed.

6. `string` consists of a list of ASCII characters, octal or hex bytes, or special symbols. You must enclose the entire sequence of characters in quotation marks. An octal byte consists of a backslash character followed by the octal value. For example, octal 40 is represented by `\040`. A hex byte consists of a backslash character and a character x followed by the hex value. For example, hex 40 is represented by `\x40`. The two special symbols, `\r` for a carriage return character and `\n` for a linefeed character, are a more convenient method for inserting the carriage return and linefeed characters into the string as shown in this example: `"F3R5T1\r\n"`. Because the carriage return can be represented equally well in hex, `\xD` and `\r` are equivalent strings.

7. Valid options for `buffer` are 0, 1, 2, and 3, which denote `off`, `ascii`, `full`, and `brief`, respectively.

Set (Select Device or Board)

Use the `set` command to change which device you are communicating with.

```
    dev1:   set plotter

    plotter:
```

This example assumes that you used `ibconf` to define the name `plotter` and that you had already called `ibfind plotter` during an earlier `ibic` session.

The following example summarizes the use of `ibfind` and `set` in a typical program.

```
 : ibfind dev1
id=32260

dev1:   ibfind plotter
id=32261


plotter:   ibwrt "F3T7G0"
[0100] (cmpl)
count: 6

plotter:    set dev1

dev1:   ibwrt "X7Y39G0"
[0100] (cmpl)
count: 7

dev1:
```

When you open a device or board using ibfind, you can use the auxiliary function set to select the opened device or board. set changes the prompt to the new symbolic name. You can also use set to switch between NI-488 mode and NI-488.2 mode.

The argument udname represents any of the symbolic device or board names recognized by the driver. The default names are gpib0, gpib1, gpib2, and gpib3 for boards, and dev1 through dev32 for devices (unless you have changed the device names using ibconf).

Help (Display Help Information)

The help function displays a menu of topics to choose from, where each topic lists relevant functions or codes. For example, the topic list1 lists all NI-488 functions. The status topic lists all status word (ibsta) codes.

Help also gives information about the syntax of functions.

! (Repeat Previous Function)

The ! function repeats the most recent function executed. The following example issues an ibsic command, then repeats that same command as follows.

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

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

- (Turn OFF Display) and + (Turn ON Display)

The – function causes the bytes received *not* to be displayed on the screen. This function is useful when you want to repeat any I/O function quickly without waiting for screen output to be displayed.

The + function causes the display to be restored.

The following example shows how the – and + functions are used. Twenty-four consecutive letters of the alphabet are read from a device using three ibrd calls.

```
    dev1:     ibrd 8
    [2100] (end cmpl)
    count: 8
    61 62 63 64 65 66 67 68        a b c d e f g h

    dev1:    -

    dev1:     ibrd 8
    [2100] (end cmpl)
    count: 8
    buffer: (off)

    dev1:    +

    dev1:     ibrd 8
    [2100] (end cmpl)
    count: 8
    71 72 73 74 75 76 77 78        q r s t u v w x
```

n* (Repeat Function n Times)

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

```
    printer:    5*ibwrt "Hello"
```

In the following example, Hello is first sent 20 more times, then 10 more times.

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

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

$ (Execute Indirect File)

The $ function reads a specified file and executes the ibic functions in sequence as if they were entered in that order from the keyboard. The following example executes the ibic functions listed in the file usrfile.

```
    gpib0:    $ usrfile
```

The following example repeats the operation three times.

```
    gpib0:    3*$ usrfile
```

The display mode, in effect before this function was executed, is restored afterward but may be changed by functions in the indirect file.

Print (Display the ASCII String)

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

```
dev1:  print "hello"
hello


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

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

ibic Examples

This section presents examples for using NI-488.2 routines and NI-488 board functions and device functions in `ibic`.

NI-488.2 Routines Example

This section shows how you might use `ibic` to test a sequence of NI-488.2 routines.

1. Use the `set` command to set up `ibic` for NI-488.2 calls.

```
: set 488.2

488.2 (0):
```

2. Send the interface clear message (IFC) to all devices. IFC clears the bus.

```
488.2 (0):  SendIFC
[0130] (cmpl cic atn)
```

3. Clear the device. The device is assumed to be on the GPIB bus at primary address 2.

```
488.2 (0):  DevClear 2
[0138] (cmpl cic atn tacs)
 count: 4
```

4. Write the routine, range, and trigger source information to a digital voltmeter.

```
488.2 (0):  Send 2 "F3R7T3" DABend
[0128] (cmpl cic tacs)
count: 6
```

5. Trigger the device.

```
488.2 (0):  Trigger 2
[0138] (cmpl cic atn tacs)
count: 4
```

6. Wait for the meter to request service (by asserting the SRQ bus line), then
    read the status byte of the meter.

```
488.2 (0):    WaitSRQ
[1138] (srqi cmpl cic atn tacs)
SRQ line is asserted


488.2 (0):    ReadStatusByte 2
[0134] (cmpl cic atn lacs)
Poll: 2 => 0x0040 (decimal : 64)
```

7. Read the data from the meter.

```
488.2 (0):    Receive 2 20 STOPend
[2124] (end cmpl cic lacs)
count: 20
0d 0a 4e 44 43 56 2d 30        • • N D C V – 0
30 30 2e 30 30 34 37 45        0 0 . 0 0 4 7 E
2b 30 0d 0a                    + 0 • •
```

NI-488 Device Functions Example

This section shows how you might use ibic to test a sequence of NI-488
device function calls.

1.   Find the device name that was given to the device in the ibconf program.

```
:  ibfind dvm
id=32259


dvm:
```

2.   Clear the device.

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

3.   Write the function, range, and trigger source instructions to the DVM.

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

4.   Trigger the device.

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

5.   Wait for a timeout or for DVM to request service. If the current timeout
      limit is too short, use ibtmo to change it.

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

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

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

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

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

8. Terminate the ibic program.

```
    dvm: e
```

NI-488 Board Functions Example

This section shows how you might use ibic to test a sequence of NI-488 board function calls.

1. Begin by opening an interface board.

```
: ibfind gpib0
id=32006


gpib0:
```

2. Send IFC to all devices. IFC clears the bus and asserts attention (ATN) on the bus.

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

3. Turn on the remote enable signal (REN).

```
gpib0:   ibsre 1
[0130] (cmpl cic atn)
previous value: 0
```

4. Set up the addressing for the device to listen and the computer to talk. The
    question mark (?) and underscore (_) characters represent the unlisten
    (UNL) and untalk (UNT) commands, respectively. These are sent in
    ibcmd to reset the bus addressing. The @ character represents the talk
    address of the GPIB board. The ! character represents the listen address of
    the device, which in this case is at GPIB primary address 1.

```
gpib0:  ibcmd "?_@!"
[0138] (cmpl cic atn tacs)
count: 4
```

5. Write the function, range, and trigger source instructions to the DVM. Be
    sure an error has not occurred before proceeding.

```
gpib0:   ibwrt "F3R7T3"
[0128] (cmpl cic tacs)
count: 6
```

6. Send the group execute trigger message (GET) to trigger a measurement
    reading. The GET message is represented by the hex value 8.

```
gpib0:    ibcmd "\x08"
[0138] (cmpl cic atn tacs)
count: 1
```

7. Wait for the DVM to set SRQ or wait for a timeout. If the current timeout
    limit is too short, use ibtmo to change it.

```
gpib0:    ibwait (TIMO SRQI)
[1138] (srqi cmpl cic atn tacs)
```

8. Set up the device for a serial poll. The question mark (?) and underscore
    (_) characters represent the unlisten (UNL) and untalk (UNT) characters,
    respectively, and reset the bus addressing. The underscore (_) represents
    the listen address of the Controller. The hex value 18 represents the serial
    poll enable function, while A represents the talk address of the device.

```
gpib0:    ibcmd "?_ \x18A"
[1174] (srqi cmpl rem cic atn lacs)
count: 5
```

9. Read the status byte. The status byte returned may vary, depending on the
    device used.

```
gpib0:     ibrd 1
[0164] (cmpl rem cic lacs)
count: 1
50          P
```

10. Complete the serial poll by sending the serial poll disable message (SPD).
    The hex value 19 represents the serial poll disable function.

```
gpib0:     ibcmd "\x19"
[0174] (cmpl rem cic atn lacs)
count: 1
```

11. Read the measurement. The DVM and the computer are still addressed to
    talk and to listen.

```
gpib0:     ibrd 20
[2164] (end cmpl rem cic lacs)
count: 20
0d 0a 4e 44 43 56 2d 30       • • N D C V - 0
30 30 2e 30 30 34 37 45       0 0 . 0 0 4 7 E
2b 30 0d 0a                   + 0 • •
```

12. Terminate the ibic program.

gpib0: *e*

Chapter 6
GPIB Programming Techniques

_____

This chapter discusses the following GPIB topics: data transfer termination
methods, waiting for GPIB conditions, device-level calls and bus management,
serial polling and SRQ servicing, and parallel polling.

Termination of Data Transfers

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

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

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

•   A 7-bit or 8-bit EOS byte.

•   The EOS comparison method, which indicates whether the EOS byte has 7
    or 8 significant bits. For a 7-bit EOS byte, the high bit of the EOS byte is
    ignored.

•   The EOS write method. If the EOS write method is enabled, the
    NI-488.2M driver automatically asserts the GPIB EOI line when the EOS
    byte is written to the GPIB. If the buffer passed into an ibwrt call
    contains five occurrences of the EOS byte, the EOI line is asserted as each
    of the five EOS bytes is 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).

- The EOS read method. If the EOS read method is enabled, the NI-488.2M 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 has been reached. If the EOS read method is disabled, `ibrd` calls terminate only when the GPIB EOI line is asserted or the specified count has been read.

You can use the `ibconfig` function to determine whether the GPIB EOI line was asserted when the EOS byte was read in. Use the `IbcEndBitIsNormal` option to configure the software to report only the END bit in `ibsta` when the GPIB EOI line is asserted. By default, the NI-488.2M driver reports END in `ibsta` when either the EOS byte is read in or the EOI line is asserted during a read.

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 0, it immediately updates `ibsta` and returns. If you want to use `ibwait` to wait for one or more events to occur, then pass a wait mask to the function. The wait mask should always include the TIMO event; otherwise, your application is suspended indefinitely until one of the wait mask events occurs.

Device-Level Calls and Bus Management

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

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

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

   1. The GPIB interface board asserts the SRQ line.

   2. The current CIC serial polls the board.

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

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

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

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 device on the bus in order 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 is set or when a manufacturer-defined condition occurs.

Automatic Serial Polling

You can enable automatic serial polling if you want your application to conduct a serial poll automatically any time the SRQ line is asserted. The autopolling procedure occurs as follows.

1. Use the configuration utility, `ibconf`, or the configuration function, `ibconfig` with option `IbcAUTOPOLL`. (Autopolling is enabled by default.)

2. When the SRQ line is asserted, the driver automatically serial polls the devices.

3. Each positive serial poll response (bit 6 or hex 40 is set) is stored in a queue associated with the device that sent it. If the poll has a positive response, the RQS bit of the device status word, `ibsta`, is set.

4. The polling continues until SRQ is unasserted or an error condition is detected.

5. The RQS bit of the device status word remains set as long as the autopoll response queue is not empty. The RQS bit is cleared when the autopoll response queue becomes empty.

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

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

If no device responds positively to the serial poll or if SRQ remains in effect because of a faulty instrument or cable, a GPIB system error occurs. In this case, the ESRQ error is reported to you if and when you issue an `ibwait` call with the RQS bit included in the wait mask. Aside from the difficulty caused by ESRQ in waiting for RQS, the error has no detrimental effects on other GPIB operations.

Autopolling and the Stuck SRQ State

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

If the driver serial polls all devices and the SRQ line is still asserted, a *stuck SRQ* state is in effect. If this happens during an `ibwait` for RQS, the driver reports the ESRQ error. If the *stuck SRQ* state happens, no further polls are attempted until either an `ibwait` for RQS or an `ibrsp` for the device whose serial poll queue is full is made. When either `ibwait` or `ibrsp` for the device whose serial poll is full is issued, the *stuck SRQ* state is terminated and a new set of serial polls is attempted.

Autopolling and Interrupts

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

If autopolling is enabled and interrupts are disabled, you can use autopolling only during a device-level `ibwait` for RQS or immediately after a device-level NI-488 function is completed and before control is returned to the application program.

SRQ and Serial Polling with NI-488 Device Functions

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

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

```
#include "decl.h"

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

      ibwait ( DeviceHandle, TIMO | RQS );

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

SRQ and Serial Polling with NI-488.2 Routines

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

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

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

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

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

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

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

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

Example 1

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

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

   WaitSRQ (0, &WaitResult);

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

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

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

  return;
}
```

Example 2

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

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

  WaitSRQ (0, &WaitResult);

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

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

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

  return;
}
```

Parallel Polling

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

Implementing a Parallel Poll

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

Parallel Polling with NI-488.2 Routines

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

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

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

    /* The following command clears the GPIB.  */

    SendIFC(0);

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

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

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

```
/* The second step performs the parallel poll and
   stores the response in response.                  */

PPoll(0, &response);

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

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

3. Unconfigure the device for parallel polling using the `PPollUnconfig` routine.

```
/* The third step disables parallel polling for
   device 3.  Notice that the NOADDR constant
   must appear at the end of the array to
   signal the end of the address list.  If
   NOADDR is the only value in the array, ALL
   devices are sent the parallel poll disable
   message.                                          */

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

Parallel Polling with NI-488 Functions

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

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

`ibppc` requires an 8-bit value to designate the data line number, the `ist` sense, and whether 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 S is 0 if the device is to assert the assigned data line when `ist` = 0.

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

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

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

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

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

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

ibppc(dev, 0x66);
```

If the GPIB interface board configures itself for a parallel poll, you should still use the `ibppc` function. Pass the board index or a board unit descriptor value as the first argument in `ibppc`. In addition, if the individual status bit (`ist`) of the board needs to be changed, use the `ibist` function. In the following example, the GPIB board is to configure itself to participate in a parallel poll. It asserts DIO5 when `ist` = 1 if a parallel poll is conducted.

```
/*Board parallel poll configuration example*/

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

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

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

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

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

Chapter 7
ibconf—Interface Bus Configuration Utility

_____

This chapter contains a description of `ibconf`, the software configuration program you can use to configure the NI-488.2M software.

Overview

The `ibconf` utility is a screen-oriented, interactive program you can use to view or modify the configuration parameters of your GPIB interface boards and the GPIB devices connected to them.

The `ibconf` utility can read in and display configuration parameters for the driver file on disk, the driver resident in memory, or the configuration data file, depending on which you select. You can then save the changes to any or all of these files.

Instead of using `ibconf`, you can configure your driver dynamically by using the `ibconfig` function to alter any board or device characteristic while your program is running. If you use dynamic configuration, you do not need to run `ibconf` before you start your application. Also, you can run your application on any computer with the appropriate NI-488.2M software regardless of its configuration, because the application configures the driver as necessary.

Note: *Dynamic configuration is the preferred method of GPIB application development*.

Starting ibconf

The `ibconf` utility is included on the distribution disk with your NI-488.2M software. After installation, `ibconf` resides in your GPIB directory. To run `ibconf`, change to your GPIB directory and enter the following command:

```
ibconf [-h[elp]] [-c] [-m] [-e]
```

where `-h` or `-help` causes the command format to be printed to the screen; `-c` enables color support (default for color monitors); `-m` sets monochrome override; and `-e` enables *expert* mode, disabling `ibconf` warning messages.

Levels of ibconf

The `ibconf` utility operates at the following four levels:

*   Input Selection level—determines which configuration file `ibconf` reads.

*   Map level—depicts the device connections as defined in the driver.

*   Description level—consists of screens describing the individual boards and devices that make up the system.

*   Output Selection level—determines which configuration files are to be updated.

The four levels are described in detail in the following sections.

Input Selection Level

Figure 7-1 shows the screen of the input selection level in `ibconf`.

```
┌──────────────────────────────────────────────────────────────────┐
│ National Instruments │ Source Configuration File │   IBM AT        │
├──────────────────────────────────────────────────────────────────┤
│                              The source file refers to the file    │
│                              from which IBCONF will obtain GPIB     │
│ Configuration Source:        driver configuration data.            │
│                                                                    │
│ Source file type ............↔ DRIVER   Source file type:          │
│   File: gpib.sys             This specifies whether configuration  │
│                              data is to be obtained from the       │
│                              memory-resident driver, from a        │
│                              driver file on the disk, or from a    │
│                              data file created by IBCONF.          │
│                                                                    │
│                              File:                                 │
│                              This is the name of the driver file   │
│                              or a data file on disk.  This field   │
│                              is applicable only if MEM-RES has not │
│                              been specified as source file type.   │
│                              Full path name of the file should be  │
│                              given.  The defaults for the driver   │
│                              file on disk and data file are        │
│                              gpib.sys and CONFIL.CF respectively.  │
├──────────────────────────────────────────────────────────────────┤
│ F1: Help              F6: Reset Value              F9: Continue     │
└──────────────────────────────────────────────────────────────────┘
```

Figure 7-1. Input Selection Level of ibconf

In the input selection level, you can select a source file that contains configuration data. When you select a source configuration file, `ibconf` obtains configuration settings from that source file. The settings are displayed and you can then edit them.

The three types of configuration files are the driver residing on your hard disk, the memory-resident driver, and the data file.

• Driver—an NI-488.2M driver file for OS/2, such as `gpib.sys`, that you copy from the distribution disk to your hard disk. Changes to a driver file do not affect your application until the system is restarted.

    Caution: ***Do not modify the master copy of*** `gpib.sys` ***on the distribution disk. Make sure your distribution disk is write protected***.

• Memory-resident driver—the NI-488.2M driver for OS/2 that is loaded and resident in memory after system startup. Changes in the memory-resident driver take effect for a given board or device only after all current processes have closed that board or device and a new open is performed.

Note: ***You cannot change device names, board interrupt levels, or the*** Use this GPIB Interface ***setting state in the memory-resident driver. To change these settings, you must change them by using*** `ibconf` ***(in the driver residing on your hard disk), then reboot the system***.

• Data file—a binary file that stores configuration data. To load a data file to the driver or the memory-resident driver, run `ibconf` and select the data file as the source.

After you select a source configuration file, continue to the next level, the map level. If you want to read a configuration from another source file, you must exit and restart `ibconf`.

Map Level

Figure 7-2 shows the map level of `ibconf`.

```
┌─────────────────────┬──────────────────────────────┬─────────────────┐
│ Driver Version 2.2  │   Device Map for Board GPIB0  │     IBM AT      │
├─────────────────────┴──────────────────────────────┴─────────────────┤
│                              * Use the cursor keys to select board/devices. │
│            ┌─────────┐       * Use PgUp/PgDn for maps of other boards.       │
│          ──┤ GPIB0   │       * Use the function keys to select an action.    │
│            └─·······─┘                                                 │
│                                                                       │
│        ──┤0│ DEV1      ──┤0│ DEV9      │1│ DEV17      │1│ DEV25       │
│        ──┤0│ DEV2      ──┤0│ DEV10     │1│ DEV18      │1│ DEV26       │
│        ──┤0│ DEV3      ──┤0│ DEV11     │1│ DEV19      │1│ DEV27       │
│        ──┤0│ DEV4      ──┤0│ DEV12     │1│ DEV20      │1│ DEV28       │
│        ──┤0│ DEV5      ──┤0│ DEV13     │1│ DEV21      │1│ DEV29       │
│        ──┤0│ DEV6      ──┤0│ DEV14     │1│ DEV22      │1│ DEV30       │
│        ──┤0│ DEV7      ──┤0│ DEV15     │1│ DEV23      │1│ DEV31       │
│        ──┤0│ DEV8      ──┤0│ DEV16     │1│ DEV24      │1│ DEV32       │
│                                                                       │
│                                                                       │
├───────────────────────────────────┬───────────────────────────────────┤
│                                                                       │
├─────────────────────────────────────────────────────────────────────┤
│F1:Help      F4:Rename     F5:(Dis)connect     F8:Edit     F9/Esc:Exit │
└─────────────────────────────────────────────────────────────────────┘
```

Figure 7-2. Map Level of ibconf

The map-level screen of `ibconf` displays the names of all devices controlled by the driver. It also indicates which devices, if any, are accessed through each interface board. You can move around the map by using the cursor control keys. For your convenience, cursor control keys and function keys are defined at the bottom of your computer screen.

The following options are available at the map level:

- Device map of the boards

- Help

- Rename

- (Dis)connect

- Edit

- Exit

The following sections describe the options available in the map level of `ibconf`.

Device Map of the Boards

Use <PageUp> or <PageDown> to toggle between the display maps for the different GPIB interface boards. These boards are referred to as access boards. The maps show which devices are assigned to each board. By default, an equal number of devices are attached to each GPIB interface board.

Help

Use <F1> to access the comprehensive, online help feature of ibconf. The help information describes the functions and common terms associated with the map level of ibconf.

Rename

Use <F4> to rename a device. Move to the device you want to rename by using the cursor control keys. Press <F4>, then enter the new name of the device. You may use up to seven letters to name your device, and you may use uppercase or lowercase letters. The following restrictions apply when naming a device:

- Extensions (.xxx) are not allowed.

- As specified by OS/2, the device name cannot have the following invalid characters (ASCII characters less than hex 21):

  ```
  .      "     /        \       [      ]        :
  |      <     >        +       =      ;        ,
  ```

- Do *not* use the reserved names con or nul for your device.

- Do *not* give GPIB device names the same names as files, directories, or subdirectories. If you name a device pltr and your file system contains the file pltr.dat or a subdirectory pltr, a conflict results.

- You cannot rename the access boards gpib0, gpib1, gpib2, and gpib3.

If you have pressed <F4 > and have then decided not to rename the device, press <F4> again to abort the function.

(Dis)connect

Use <F5> to connect or disconnect a device to or from a particular access board. Move the cursor to the device that is to be connected or disconnected by using the cursor control keys and then pressing <F5>.

Edit

Use <F8> to edit or examine the characteristics of a particular board or device. Move to the board or device that you want to edit by using the cursor control keys and pressing <F8>. This places you in the description level of `ibconf` and lists all the characteristics for the particular board or device that you want to edit. To exit edit, use <F9> or <Esc>.

Exit

Use <F9> or <Esc> to exit `ibconf`. If you have made changes and have pressed <F9> to exit, `ibconf` displays `Save configuration?`. Type `y` (yes) to save the changes or `n` (no) to discard the changes. For more information, refer to the *Exiting ibconf* section, which is located later in this chapter.

If you have pressed <F9> and have then decided not to exit, press <F9> again to abort the function.

Description Level

The description-level screens of `ibconf` display current values, such
as addressing and timeout information, for device or board settings. See Figure
7-3.

Figure 7-3. Description Level of ibconf

You can access the description-level screens from the map level of `ibconf` by
selecting a board or device and pressing <F8>. You can use the <Up>, <Down>,
<PageUp>, and <PageDown> cursor keys to select a characteristic. For your
convenience, cursor control keys and function keys are defined at the bottom of
your computer screen.

Selecting the configuration settings for each device and board customizes the
communications and other options to be used with that board or device. These
settings are the characteristics used by the access board when device functions
are used to program the device or when board functions are used to program the
board.

The following options are available at the description level:

•　Change Characteristics

•　Next Board/Device

- Help

- Reset Value

- Return to Map

The following sections describe the options available in the description level of `ibconf`.

Change Characteristics

To change a specific characteristic of a device or a board, first move your cursor to the characteristic. If the double arrow symbol appears to the left of the input field, use the left/right arrow keys to select between different options. If a blinking cursor appears in the field, input the new value directly from the keyboard. Instructions on the right half of the screen inform you about the options for the characteristic.

Next Board/Device

If you are viewing the characteristics of a board, use <Ctrl-PageUp> and <Ctrl-PageDown> to move to the characteristics screen for the previous or next board. If you are viewing the characteristics of a device, use <Ctrl-PageUp> and <Ctrl-PageDown> to move to the characteristics screen for the previous or next device.

Help

Use <F1> to access the comprehensive, online help feature of `ibconf`. The help information describes the functions and common terms associated with the description level of `ibconf`.

Reset Value

Use <F6> to reset the current characteristic to the value it contained when it was last modified.

Return to Map

Use <F9> or <Esc> at the description level to return to the map level of `ibconf`.

Output Selection Level

Figure 7-4 shows the output selection level of ibconf.

```
┌─────────────────────┬──────────────────────────────┬──────────────────┐
│ Driver Version 2.2  │ Destination Configuration Files │     IBM AT      │
├─────────────────────┴──────────────────────────────┴──────────────────┤
│                                    Copy changes to memory-res:         │
│                                    Configuration changes are written to │
│ Configuration Destination(s):      the temporary memory-resident copy  │
│                                    of the driver. The changes will not  │
│ Copy changes to memory-res? ..↔ No have any effect on open boards or    │
│ Copy changes to driver on disk? Yes devices. The changes take effect    │
│   File: gpib.sys                   for a board/device only after a new  │
│ Copy changes to data file? .... No open of board/device is performed.  │
│   File: Not applicable                                                  │
│                                    Three parameters, device names,      │
│                                    board interrupt levels and "IN USE"  │
│                                    status of boards cannot be changed   │
│                                    in the memory resident driver.       │
│                                                                         │
│                                                                         │
│                                                                         │
│                                                                         │
├─────────────────────────────────────────────────────────────────────── │
│ F1: Help            F6: Reset Value              F9/Esc: Continue       │
└─────────────────────────────────────────────────────────────────────────┘
```

Figure 7-4. Output Selection Level of ibconf

In the output selection level, you can select the configuration files that should be updated by ibconf. To create or update a data file, respond yes to copy changes to the data file. The data file confil.cf (the default name) is then created or opened and the new configuration is stored into it.

If you select yes to copy changes to the memory resident driver, ibconf updates the driver that is currently resident in memory. If you select yes to copy changes to drivers on disk, ibconf updates the driver file on disk.

Board and Device Configuration Options

To view detailed information about each characteristic, position the cursor in the field for that characteristic. For information on characteristics specific to a given driver, check the getting-started manual that came with your interface board. The following paragraphs describe the options available in ibconf for the NI-488.2M software for OS/2.

Primary GPIB Address

All devices and boards must be assigned a unique primary address in the range hex 00 to hex 1E (0 to 30 decimal). The default primary address of all GPIB boards is 0.

The GPIB primary address of any device is set within the device, either with hardware switches or a software program. The address set within the device must correspond to the address in the memory-resident driver. In the NI-488.2M driver for OS/2, the default primary addresses of `dev1` through `dev32` are 1 through 32, respectively. Refer to the device documentation for instructions about setting the device address. GPIB boards do *not* have hardware switches to select the GPIB address.

The primary GPIB address is used to compute the talk and listen addresses of devices and boards. The NI-488.2M driver automatically forms a listen address by adding hex 20 to the primary address and a talk address by adding hex 40 to the primary address. For example, a primary address of hex 10 has a listen address of hex 30 and a talk address of hex 50.

Secondary GPIB Address

Any device or board using extended addressing must be assigned a secondary address in the range hex 60 to hex 7E (96 to 126 decimal), or the option `NONE` may be selected to disable secondary addressing.

As with primary addressing, the secondary GPIB address of a device is set within that device, either with hardware switches or a software program. The address set within the device must correspond to the address in the memory-resident driver. Refer to the device documentation for instructions about secondary addressing. The default setting for this characteristic is `NONE` for all boards and devices.

Timeout Setting

The timeout value is the approximate length of time that GPIB functions wait for data to be transferred or commands to be sent. It is also the length of time that the `ibwait` function waits for an event before returning, if the TIMO bit is set in the event mask. For example, if the SRQI bit and TIMO bit in the event mask are passed to the `ibwait` function and no SRQ is detected, the function timeouts after the duration given by the timeout setting value. The default option for this characteristic is 10 s.

Terminate Read on EOS

Some devices send an EOS byte signaling the last byte of a data message.
A `yes` response in this field causes the GPIB board to terminate a read operation
when it receives the EOS byte. The default option for this characteristic is `no`.

Set EOI with EOS on Writes

A `yes` response in this field causes the GPIB board to assert the EOI line when
the EOS byte is detected on a write operation. The default option for this
characteristic is `no`.

Type of Compare on EOS

This field specifies the type of comparison to be made with the EOS byte. You
may state whether all eight bits are to be compared or just the seven least
significant bits (ASCII or ISO [International Standards Organization] format).
The default option for this characteristic is `7-bit`.

Note: *This field is valid only if a* `yes` *response was given for either the Set
EOI with EOS on Write field or the Terminate Read on EOS field.*

EOS Byte

Some devices can be programmed to terminate a read operation when a selected
character is detected. A linefeed character (hex 0A) is a common EOS byte.
The default option for this characteristic is 00H.

Note: *The driver does not automatically append an EOS byte to the end of
data strings on write operations. You must explicitly include this byte
in your data string. The EOS byte designation informs the driver of
its value so that I/O can terminate correctly*.

Send EOI at End of Write

Some devices, as Listeners, require that the Talker terminate a data message by asserting the EOI line with the last byte. A `yes` response causes the GPIB board to assert the EOI line on the last data byte. The default option for this characteristic is `yes`.

GPIB-Specific Errors

Boards and devices have a switch that specifies whether the NI-488.2M driver returns OS/2 device errors or GPIB device errors when API `IOCtl` functions are used. A `no` response causes the OS/2 API functions (except the status query function) to return OS/2 device errors. If this option is set to `yes`, GPIB-specific errors are returned on all API `IOCtl` functions. The default option for this characteristic is `yes` .

System Controller (Board Characteristic Only)

This field appears on the board characteristics screen only. The System Controller in a GPIB system is the device that maintains ultimate control over the bus. In some situations, such as a network of computers linked by the GPIB, another device may be System Controller and the GPIB board should be designated as *not* System Controller. A `no` response would designate *not* System Controller and a `yes` response would designate System Controller capability. The GPIB board is usually designated as System Controller. The default option for this characteristic is `yes`.

Note: ***You should not have more than one designated System Controller in any GPIB system***.

Assert REN when SC (Board Characteristic Only)

A `yes` response to this field causes Remote Enable (REN) to be asserted automatically whenever the board is placed online, if that the board has System Controller capability. If you give a `no` response, an explicit call to `ibsre` is required to assert REN. The default option for this characteristic is `no`.

Enable Auto Serial Polling (Board Characteristic Only)

This option enables or disables automatic serial polls of devices when the GPIB Service Request (SRQ) line is asserted. Positive poll responses are stored following the polls and can be read with the `ibrsp` device function. Normally, this feature does not conflict with devices that conform to the IEEE 488 standard. If there is a conflict with a device, a `no` response for this field disables this feature. The default option for this characteristic is `yes`.

Enable CIC Protocol (Board Characteristic Only)

If a device-level NI-488 call is made after control has been passed to another device, enabling this protocol causes the board to assert SRQ with a Serial Poll status byte of hex 42. The current Controller must recognize that the board wants to regain control. If the current Controller passes control back to the board, the device call proceeds as intended. If control is not passed within the timeout period, the ECIC error is returned. If the CIC protocol is disabled, ECIC is returned immediately if a device call is made after control has been passed. The default option for this characteristic is `no`.

Bus Timing (Board Characteristic Only)

This field specifies the T1 delay of the source handshake capability for the board. This delay determines the minimum amount of time, after the data is placed on the bus, that the board may assert DAV during a write or command operation. If the total length of the GPIB cable length in the system is less than 15 m, the value of 350 ns is appropriate. Other factors might affect the choice of the T1 delay, although they are unlikely to affect your system setup. Refer to the IEEE 488.1 standard, Section 5.2, for more information about these other factors. The default for this option is 500 ns.

Parallel Poll Duration (Board Characteristic Only)

This field specifies the length of time the driver waits when conducting a parallel poll. For a normal bus configuration (the Controller and devices on the same bus), use the default duration of 2 μs. If you are using a GPIB bus extender in transparent parallel poll mode, you should increase the poll duration to 10 μs so that the bus extender can operate transparently to your applications.

Use This GPIB Interface (Board Characteristic Only)

If you do not want the driver to try to access a board at the selected base address (because you do not have a board in the system), select `no` for this option. When this field is set to `no`, the driver returns the EDVR error as soon as a program tries to access the board. By default, access board `gpib0` is enabled, and `gpib1`, `gpib2`, and `gpib3` are disabled.

Base I/O Address (Board Characteristic Only)

This field specifies the I/O address of the GPIB board. It must be set to the same value as the base I/O address setting for the GPIB board. Setting the base I/O address level is explained in the getting-started manual that you received with your GPIB interface board.

Note: ***On Micro Channel systems, this field is Read Only and you can change the I/O address only by booting the reference diskette***.

DMA Channel (Board Characteristic Only)

This field specifies the DMA channel used by the GPIB board. It must be set to the same value as the DMA channel (arbitration level for Micro Channel systems) setting for the GPIB board. Setting the DMA channel is explained in the getting-started manual that you received with your GPIB interface board.

Note: ***On Micro Channel systems, you can change the DMA channel only by booting the reference diskette. However, you can enable or disable the use of DMA with the*** `ibconf` ***utility***.

Interrupt Jumper Setting (Board Characteristic Only)

This field specifies the interrupt line used by the GPIB board. It must be set to the same value as the interrupt level setting for the GPIB board. Setting the interrupt level is explained in the getting-started manual that you received with your GPIB board.

Note: *On Micro Channel systems, you can change the interrupt level only by booting the reference diskette. However, you can enable or disable the use of interrupts with the* `ibconf` *utility.*

DMA Transfer Mode (Board Characteristic Only)

This field specifies the DMA transfer mode of the GPIB board. Nearly all PC-compatible computers can use Demand Mode DMA, which is the fastest DMA mode. Certain newer machines, however, cannot use Demand Mode DMA; these machines should use Single Cycle DMA. The default option for this characteristic is `Demand Mode DMA`.

Note: *This characteristic is not applicable to Micro Channel systems.*

Serial Poll Timeout (Device Characteristic Only)

This timeout value controls the length of time the driver waits for a serial poll response from a device. The IEEE 488 standard does not specify the length of time a Controller should wait for the response byte. The default value of 1 s should work for most devices. If you have problems with serial polls, try using a longer timeout value.

Enable Repeat Addressing (Device Characteristic Only)

Normally, devices are not addressed each time a read or write operation is performed. If `no` is selected, read or write operations do not readdress the selected device if the same operation was just performed with that device. This saves some time when you have several GPIB operations to perform after repeat addressing. But it might be a problem for some older IEEE 488 devices that require their GPIB address to be sent with each I/O operation. Select `yes` to enable repeat addressing in such a situation. The default option for this characteristic is `no`.

Default Configurations in ibconf

This section lists the default configuration values of the NI-488.2M driver.

- Thirty-two devices with symbolic names `dev1` through `dev32`.

- Four access boards with symbolic names `gpib0`, `gpib1`, `gpib2`, and `gpib3`. You cannot change the access board names.

- Access board `gpib0` is enabled. `gpib1`, `gpib2`, and `gpib3` are disabled.

- The GPIB addresses of the first 16 devices are the same as the device number. For example, `dev1` is at address 1. These 16 devices are assigned to the access board `gpib0`.

- The remaining 16 devices (that is, devices 17 through 32) are assigned to the access board `gpib1`. Their GPIB addresses range from 1 through 16, respectively. For example, `dev17` is at address 1.

- Each GPIB interface board is System Controller for its independent bus and has a GPIB address of 0.

- The END message is sent with the last byte of each data message to a device. No EOS character is recognized.

- The time limit on I/O and wait function calls is set for 10 s.

- Each GPIB board and device is set to perform I/O transfers using DMA.

- Automatic serial polling is enabled.

- Each GPIB board and device is set to return GPIB-specific errors while executing API-style calls.

- At the end of each NI-488.2 routine, the NI-488.2M driver leaves the bus in its currently addressed state (IEEE 488.2 standard).

- The time limit for a serial poll is set for 1 s.

Exiting ibconf

After you make any changes in ibconf, use <F9> or<Esc> to exit the program. If you have made changes to device/board characteristics, device names, or device connections, the program first displays the prompt Save configuration? before exiting. Type a y (yes) to save changes or n (no) to delete changes.

Before exiting, if not in expert mode, ibconf automatically checks for situations, such as the following, that can cause problems:

• GPIB addressing conflicts between a device and its access board.

• GPIB boards not present in the host machine at the specified address.

• Timeouts disabled on a device or board.

• Interrupt level of a board is the same as another board.

If ibconf detects any problems, it notifies you and gives you the option of either re-entering or exiting ibconf. To disable automatic checking, use expert mode by entering the following command when you start ibconf:

ibconf -e

Appendix A
Status Word Conditions

This appendix gives a detailed description of the conditions reported in the status word, ibsta. For information about how to use ibsta in your application program, refer to Chapter 3, *Developing Your Application*.

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

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

The following table lists the status word bits.

Table A-1. Status Word (ibsta) Layout

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

ERR (dev, brd)

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

TIMO (dev, brd)

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

END (dev, brd)

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

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

SRQI (brd)

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

RQS (dev)

RQS appears in the status word only after a device-level call. RQS indicates that one or more automatic serial poll response bytes are waiting in the device's serial poll response queue. Automatic serial poll responses are not stored in the response queue unless they have bit 6 set.

An automatic serial poll occurs as a result of a call to `ibwait`, or it occurs automatically if automatic serial polling is enabled. If the serial poll response queue is not empty, `ibrsp` returns the oldest byte stored in the queue. To empty the response queue, call `ibrsp` repeatedly until RQS is no longer set in the device's status word.

CMPL (dev, brd)

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

LOK (brd)

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

REM (brd)

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

•   When REN becomes unasserted.

•   When the GPIB board as a Listener detects that the Go to Local (GTL) command has been sent either by the GPIB board or by another Controller.

•   When the `ibloc` function is called while the LOK bit is cleared in the status word.

CIC (brd)

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

ATN (brd)

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

TACS (brd)

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

LACS (brd)

LACS indicates whether the GPIB board is addressed as a Listener. LACS is set whenever the GPIB board detects 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 ibgts is called without shadow handshake.

DTAS (brd)

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

DCAS (brd)

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

Appendix B
Error Codes and Solutions

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

The following table lists the GPIB error codes.

Table B-1. GPIB Error Codes

Error Mnemonic	iberr Value	Meaning
EDVR	0	OS/2 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
EOIP	10	Asynchronous I/O in progress
ECAP	11	No capability for operation
EFSO	12	File system error
EBUS	14	GPIB bus error
ESTB	15	Serial poll status byte queue overflow
ESRQ	16	SRQ stuck in ON position
ETAB	20	Table problem

EDVR (0)

EDVR is returned when the device or board name passed in an `ibfind` call is not configured in the software. In this case, the variable `ibcntl` contains the OS/2 error code 2, *Device not found*, or error code 110, *Open failed*.

EDVR is also returned when an invalid unit descriptor is passed to any function call. In this case, the variable `ibcntl` contains the OS/2 error code 6, *Invalid handle*.

EDVR is also returned when the driver (`gpib.sys`) is not installed.

Solutions

1. Use only device or board names that are configured in the software utility `ibconf` as parameters in the `ibfind` function.

2. Use the unit descriptor returned from the `ibfind` function as the first parameter in subsequent NI-488 functions. Examine the variable after the `ibfind` and before the failing function to make sure it was not corrupted.

3. Make sure the NI-488.2M driver is installed by checking the `gpibinst.inf` file in the root directory.

4. Check the `config.sys` file in the root directory and make sure it contains the following line:

    `DEVICE=`*dir*`\gpib.sys`

    where *dir* is the directory that contains the file `gpib.sys`.

ECIC (1)

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

- Any of the NI-488.2 routines that issue GPIB command bytes: `SendCmds`, `PPoll`, `Send`, `Receive`.

- Any board-level functions that issue GPIB command bytes: `ibcmd`, `ibcmda`, `ibln`, `ibrpp`.

- `ibcac`, `ibgts`.

- Any device-level functions that affect the GPIB.

Solutions

1. Use `ibsic` or `SendIFC` to make the GPIB board become CIC on the GPIB.

2. Use `ibrsc 1` or run `ibconf` and make sure your GPIB board is configured as System Controller.

3. In multiple CIC situations, always be certain that the CIC bit appears in the status word `ibsta` before attempting these calls. If the CIC bit does not appear, you can perform an `ibwait` (for CIC) call to delay further processing until control is passed to the board.

ENOL (2)

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

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

Solutions

1.   Make sure that the GPIB address of your device matches the GPIB address of the device you want to write data to.

2.   If you are using board-level functions, make sure that your device is properly addressed to listen before writing to it by using `ibcmd` or `sendsetup`.

3.   Use the appropriate hex code in `ibcmd` to address your device.

4.   Check your cable connections and make sure at least two-thirds of your devices are powered on.

5.   Call `ibpad` (or `ibsad`, if necessary) to match the configured address to the device switch settings.

6.   Reduce the write byte count to that which is expected by the Controller.

EADR (3)

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

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

Solutions

1.   Make sure that the GPIB board is addressed correctly before calling `ibrd` or `ibwrt`.

2   Avoid calling `ibgts` except immediately after an `ibcmd` call (`ibcmd` causes ATN to be asserted).

EARG (4)

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

- `ibtmo` called with a value not in the range 0 through 17.

- `ibeos` called with meaningless bits set in the high byte of the second parameter.

- `ibpad` or `ibsad` called with invalid addresses.

- `ibppc` called with invalid parallel poll configurations.

- A board-level function made with a valid device descriptor, or a device-level function made with a board descriptor.

- An NI-488.2 routine called with an invalid address.

- `PPollConfig` called with an invalid data line or sense bit.

- `Termination` parameter in `RcvRespMsg` and `Receive` is neither `STOPend` or an 8-bit EOS character.

- `eotmode` parameter in `Send`, `SendDataBytes`, and `SendList` is not `DABend`, `NULLend`, or `NLend`.

- In `Send`, `SendDataBytes`, or `SendList` routine, `eotmode` is `DABend` and `datacnt` is 0.

Solutions

1. Make sure that the parameters passed to the NI-488 function or NI-488.2 routine are valid.

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

ESAC (5)

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

Solutions

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

EABO (6)

EABO indicates that an I/O operation has been canceled—usually because of a timeout condition. Other causes are calling `ibstop` or receiving the Device Clear message from the CIC while performing an I/O operation.

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

Solutions

1. Use the correct byte count in input functions or have the Talker use the END message to signify the end of the transfer.

2. Lengthen the timeout period for the I/O operation using `ibtmo`.

3. Make sure that you have configured your device to send data before you request data.

ENEB (7)

ENEB occurs when a GPIB board does not exist at the I/O address specified in the configuration program. This situation 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 conflict in the system with the base I/O address.

Solutions

1. Make sure a GPIB board is in your computer that is configured both in hardware and software at a free base I/O address.

2. Make sure that the `Use this GPIB Interface` field in `ibconf` is set to `Yes`.

EOIP (10)

EOIP occurs when an asynchronous I/O operation has not finished before some other call is made. During asynchronous I/O, you can use only `ibstop`, `ibwait`, and `ibonl`. If any other call is attempted, EOIP is returned.

With the asynchronous I/O calls (`ibcmda`, `ibrda`, `ibwrta`), your application can perform additional non-GPIB operations while the I/O is in progress. Once the asynchronous I/O has begun, further GPIB calls other than `ibstop`, `ibwait`, or `ibonl` are strictly limited. If a call interferes with the I/O operation in progress, it causes the driver to return EOIP.

Solutions

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

    - `ibwait` If the returned `ibsta` contains CMPL, the driver and application are resynchronized.

    - `ibstop` The I/O is canceled; the driver and application are resynchronized.

    - `ibonl`   The I/O is canceled and the interface is reset; the driver and application are resynchronized.

ECAP (11)

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

Solutions

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

EFSO (12)

EFSO results when an `ibrdf` or `ibwrtf` call encounters a problem
performing a file operation. Specifically, this error indicates that the
function is unable to open, create, seek, write, or close the file being
accessed.

Solutions

1. Make sure the filename, path, and drive that you specified are correct.

2. Make sure that the access mode of the file is correct.

3. Make sure there is enough room on the disk to hold the file.

EBUS (14)

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

Solutions

1. Verify that the instrument is operating correctly.

2. Check for loose or faulty cabling or several powered off instruments
    on the GPIB.

3. If the timeout period is too short for the driver to send command bytes,
    increase the timeout period.

ESTB (15)

ESTB occurs only during the `ibrsp` function. ESTB indicates that auto polling was not done for a device whose serial poll response queue was full. Several older status bytes are available; however, the oldest is being returned by the `ibrsp` call.

Solutions

1. Call `ibrsp` more frequently to empty the queue.

2. Disable autopolling with the `ibconfig` function or the `ibconf` utility.

ESRQ (16)

ESRQ occurs only during the `WaitSRQ` routine or `ibwait` function. ESRQ indicates that the GPIB SRQ line is stuck on. This situation can be caused by the following events:

- Usually, a device unknown to the software is asserting SRQ. Because the software does not know of this device, it can never serial poll the device and unassert SRQ.

- A GPIB bus tester or similar equipment might be forcing the SRQ line to be asserted.

- A cable problem, involving the SRQ line, might exist.

- The serial poll response queue of a device that is asserting the SRQ line is full.

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

Solutions

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

2. Call `ibrsp` for the device whose serial poll queue is full.

ETAB (20)

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

- In the case of `FindLstn`, ETAB means that the given table did not have enough room to hold all the addresses of the Listeners found.

- In the case of `FindRQS`, ETAB means that none of the devices in the given table were requesting service.

Solutions

1. Increase the size of result arrays for `FindLstn` and `FindRQS`.

Appendix C
Customer Communication

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

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

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

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

Technical Support Form

Technical support is available at any time by fax. Include the information from the configuration form in your Getting Started Manual. Use additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___)_____   Phone (___)_____

Computer brand _____

Model _____   Processor _____

Operating system _____

Speed _____MHz   RAM _____MB

Display adapter _____

Mouse _____yes   _____no

Other adapters installed_

Hard disk capacity _____MB   Brand _____

Instruments used _____

National Instruments hardware product model _____

Revision _____

Configuration _____

(continues)

National Instruments software product _____

    Version _____

    Configuration _____

The problem is _____

_____

_____

_____

_____

_____

_____

_____

List any error messages _____

_____

_____

_____

_____

_____

_____

_____

The following steps will reproduce the problem _____

_____

_____

_____

_____

_____

_____

Documentation Comment Form

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

Title: **NI-488.2M™ Software Reference Manual for OS/2**

Edition Date: **January 1995**

Part Number: **370950A-01**

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

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

(continues)

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

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

Phone (_____) _____

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

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

Glossary

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

A

Acceptor Handshake	A GPIB interface function that receives data or commands. Listeners use this function to receive data, and all devices use it to receive commands. See *Source Handshake* and *Handshake.*
Access Board	The GPIB board that controls and communicates with the devices on the bus that are attached to it.
ANSI	American National Standards Institute
API	Application Program 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.
ATN (Attention)	A GPIB line that distinguishes between commands and data messages. When ATN is asserted, bytes on the GPIB DIO lines are commands.

Automatic Serial Polling (Autopolling)	A feature of the NI-488.2M software in which serial polls are executed automatically by the driver whenever a device asserts the GPIB SRQ line.

B

Board Function	A function that operates on or otherwise pertains to one of the GPIB interface boards in the computer.

C

CIC	See *Controller-In-Charge*.
`config.sys`	An OS/2 file that contains the names of the loadable device driver or driver programs that OS/2 loads when it is started up.
Controller-In-Charge (CIC)	The device that manages the GPIB by sending interface messages to other devices.

D

DAV (Data Valid)	One of the three GPIB handshake lines. See *Handshake*.
DCL	Device Clear is the GPIB command used to reset the device or internal functions of all devices. See *SDC*.
Declaration File	A file containing definitions that must be placed at the beginning of an application program. `decl.h` is the declaration file for C programs using the NI-488 functions and NI-488.2 routines. See *Language Interface*.
Device Clear	See DCL.

Device Function	A function that operates on or otherwise pertains to a GPIB device rather than to the GPIB interface board in the computer. See *Board Function*.
DIO1 through DIO8	The GPIB lines that are used to transmit command or data bytes from one device to another.
DLL	dynamic link library
DMA (direct memory access)	High-speed data transfer between the GPIB board and memory that is not handled directly by the CPU. Not available on some systems. See *Programmed I/O*.
Driver	Device driver software installed within the operating system. Same as an OS/2-installed device driver.
DVM	digital voltmeter

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

G

GET	Group Execute Trigger is the GPIB command used to trigger a device or internal function of an addressed Listener.
Go To Local	See *GTL*.
GPIB	General Purpose Interface Bus is the common name for the communications interface system defined in ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1987.
GPIB Address	The address of a device on the GPIB, composed of a primary address (MLA and MTA) and 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.
gpib.sys	The NI-488.2M driver file name.
Group Executed Trigger	See *GET*.
GTL	Go To Local is the GPIB command used to place an addressed Listener in local (front panel) control mode.

H

Handshake	The mechanism used to transfer bytes from the Source Handshake function of one device to the Acceptor Handshake function of another device. The three GPIB lines, DAV, NRFD, and NDAC, are used in an interlocked fashion to signal the phases of the transfer, so that bytes can be sent asynchronously—for example, without a clock— at the speed of the slowest device.
hex	hexadecimal

High-Level Function	A device function that combines several rudimentary board operations into one function so that the user does not have to be concerned with bus management or other GPIB protocol matters. See *Low-Level Function*.
Hz	hertz

I

ibcnt	After each NI-488.2M I/O function, this global variable contains the actual number of bytes transmitted.
ibconf	The NI-488.2M driver configuration program.
iberr	A global variable that contains the specific error code associated with a function call that failed.
ibic	The Interface Bus Interactive Control program is used to communicate with GPIB devices, troubleshoot problems, and develop your application.
ibsta	At the end of each function call, this global variable (status word) contains status information.
IFC or Interface Clear	A GPIB line used by the System Controller to initialize the bus. See *DCL* and *SDC*.
Interface Message	A broadcast message sent from the Controller to all devices and used to manage the GPIB.
I/O (Input/Output)	In the context of this manual, the transmission of commands or messages between the computer via the GPIB board and other devices on the GPIB.

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

L

LAD or Listen Address	See *MLA*.
Language Interface	Code that enables an application program that uses NI-488.2M functions or NI-488.2M routines to access the driver.
Listen Address	See *MLA*.
Listener	A GPIB device that receives data messages from a Talker.
LLO	Local Lockout is the GPIB command used to tell all devices that they may or should ignore remote (GPIB) data messages or local (front panel) controls, depending on whether the device is in local or remote program mode.
Low-Level Function	A rudimentary board or device function that performs a single operation. See *High-Level Function*.

M

Make File	Utility that compiles and links programs.
MB	Megabytes of memory.
Memory-Resident	Resident in RAM.
MLA (My Listen Address)	A GPIB command used to address a device to be a Listener. There are 31 primary addresses.

MSA	My Secondary Address is the GPIB command used to address a device to be a Listener or a Talker when extended (two byte) addressing is used. The complete address is an 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. There are 31 primary addresses.
Multitasking	The concurrent processing of more than one program or task. OS/2 provides a multitasking environment so that multiple applications can execute at the same time.

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

Open Device or Board	One that has been enabled or placed online by a system or language open function.

P

Parallel Poll	The process of polling all configured devices at once and reading a composite poll response. See *Serial Poll*.
PIO	See *Programmed I/O*.

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

R

RAM	random-access memory
REN (Remote Enable)	A GPIB line controlled by the System Controller but used by the CIC to place devices in remote program mode.
RQS	Request Service

S

s	seconds
SDC	Selected Device Clear is the GPIB command used to reset internal or device functions of an addressed Listener. See *DCL* and *IFC*.

Serial Poll	The process of polling and reading the status byte of one device at a time. See *Parallel Poll*.
Service Request	See *SRQ*.
Source Handshake	The GPIB interface function that transmits data and commands. Talkers use this function to send data, and the Controller uses it to send commands. See *Acceptor Handshake* and *Handshake*.
SPD (Serial Poll Disable)	Serial Poll Disable is the GPIB command used to cancel an SPE command.
SPE (Serial Poll Enable)	Serial Poll Enable is the GPIB command used to enable a specific device to be polled. That device must also be addressed to talk. See *SPD*.
SRQ (Service Request)	The GPIB line that a device asserts to notify the CIC that the device needs servicing.
Status Byte	The data byte sent by a device when it is serially polled.
Status Word	See *ibsta*.
Synchronous	Refers to the relationship between the NI-488.2M driver functions and a process when executing driver functions is predictable; the process is blocked until the OS/2 driver completes the function.
System Controller	The single-designated Controller that can assert control (become CIC of the GPIB) by sending the Interface Clear (IFC) message. Other devices can become CIC only by having control passed to them.

T

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

U

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

Index

Numbers/Symbols

! (repeat previous function), ibic
  utility, 5-21
$ (execute indirect file) function,
  ibic utility, 5-23
+ (turn on display) function, ibic
  utility, 5-21 to 5-22
- (turn off display) function, ibic
  utility, 5-21 to 5-22

A

addresses
  base I/O address, 7-14
  ENEB error code, B-6
  ENOL error code, B-3 to B-4
  primary GPIB address, 7-10
  secondary GPIB address, 7-10
addressing
  enable repeat addressing, 7-15
  repeat addressing, 4-6
AllSpoll function, 6-6, 6-7, 6-8
ANSI/IEEE Standard
  488.1-1987, 1-5
ANSI/IEEE Standard
  488.2-1987, 1-5
API interface. *See* OS/2 API
  interface.
application examples
  asynchronous I/O, 2-6 to 2-7
  basic communication, 2-2
    to 2-3
      with IEEE 488.2 compliant
        devices, 2-14 to 2-15
  clearing and triggering devices,
    2-4 to 2-5
  end-of-string mode, 2-8 to 2-9
  files on distribution disk, 1-1 to
    1-2, 2-1

non-controller, 2-21 to 2-22
  parallel polls, 2-18 to 2-20
  serial polls using NI-488.2
    routines, 2-16 to 2-17
  service requests, 2-10 to 2-13
applications, debugging
  checking global variable
    status, 4-3
  communication errors, 4-6
  configuration errors, 4-4
  GPIB error codes, 4-3 to 4-4
  ibic, 4-3
  ibtest, 4-1 to 4-2
  reconfiguring NI-488.2M
    software, 4-5
  repeat addressing, 4-6
  termination method, 4-6
  timing errors, 4-5
applications, writing. *See also*
  application examples.
  choosing programming method,
    3-1 to 3-4
  compiling and linking programs
    16-bit C applications, 3-21
    32-bit C applications, 3-20
      to 3-21
  ibic for communicating with
    devices, 3-7
  NI-488 applications, 3-7 to 3-12
    clearing devices, 3-10
    configuring devices, 3-10
    items to include, 3-7
    NI-488 program shell, 3-8
    opening devices, 3-9
    placing devices
      offline, 3-12
    processing data, 3-12
    reading measurement, 3-12
    triggering devices, 3-10
      to 3-11

H

I

S

secondary GPIB address, 7-10
Send routine
    configuring instruments, 3-17
    ibic utility, 5-11 to 5-12
SendIFC routine, 3-15
serial polling
    automatic, 6-4 to 6-5
        enabling, 7-13
        interrupts and, 6-5
        procedure for, 6-4 to 6-5
        stuck SRQ state and, 6-5
    service requests
        from IEEE 488
            devices, 6-4
        from IEEE 488.2
            devices, 6-4
    SRQ and serial polling
        with NI-488 device
            functions, 6-6
        with NI-488.2 routines, 6-6
            to 6-8
    timeout value, setting, 7-15
    using NI-488.2 routines
        (example), 2-16 to 2-17
service requests
    automatic serial polling, 6-4
        to 6-5
    example, 2-10 to 2-13
    RQS status, A-3
    serial polling
        IEEE 488 device
            functions, 6-6
        IEEE 488.2 routines, 6-6
            to 6-8
    SRQI status, A-2
Set function
    description, 5-19 to 5-20
    selecting NI-488.2 function
        mode, 5-11
SRQ (service request) line

application example, 2-10
    to 2-13
description (table), 1-7
ESRQ error code, B-9
requesting service, 6-3
serial polling
    with NI-488 device
        functions, 6-6
    with NI-488.2 routines, 6-6
        to 6-8
    stuck SRQ state, 6-5
SRQI status, A-2
standards
    ANSI/IEEE Standard
        488.1-1987, 1-5
    ANSI/IEEE Standard
        488.2-1987, 1-5
status variables. *See* global
    variables.
status word variable. *See* ibsta
    (status word).
stuck SRQ condition, 6-5
System Controller
    configuring, 7-12
    definition, 1-6

T

TACS status, A-4
Talkers
    definition, 1-5
    TACS status, A-4
technical support, C-1
terminate read on EOS, 7-11
termination methods
    data transfer termination, 6-1
        to 6-2
    device termination methods, 4-6
TestSRQ routine, 3-18, 6-7
timeout setting
    configuring, 7-10
    serial polls, 7-15