NI-488® and NI-488.2™
Subroutines for NKR BASIC

August 1992 Edition

Part Number 320348-01

National Instruments Corporate Headquarters
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 794-0100
(800) 433-3488 (toll-free U.S. and Canada)
Technical support fax: (512) 794-5678

Branch Offices:
Australia 03 879 9422, Belgium 02 757 00 20, Canada 519 622 9310,
Denmark 45 76 73 22, Finland 90 524566, France 1 48 65 33 70,
Germany 089 714 50 93, Italy 02 48301892, Japan 03 3788 1921,
Netherlands 01720 45761, Norway 03 846866, Spain 91 896 0675,
Sweden 08 984970, Switzerland 056 27 00 20, U.K. 0635 523545

Limited Warranty

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

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

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

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

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

Copyright

Trademarks

Preface

This manual contains information for programming the NI-488.2 routines and the NI-488 functions in NKR BASIC. The term *NKR BASIC*, as used in this manual, refers to NKR BASIC for MS-DOS.

This manual assumes that the driver is installed and that you are familiar with the driver operation. Programming knowledge in NKR BASIC and familiarity with the compiler are also assumed.

Organization of This Manual

This manual is organized as follows:

- Chapter 1, *General Information*, lists the files relevant to programming in NKR BASIC, contains programming preparations, discusses how to use the NI-488.2 routine and NI-488 function examples, and summarizes the calls that will be explained at length in Chapter 2 and Chapter 3.

- Chapter 2, *NI-488.2 Routine Descriptions*, contains a detailed description of each NI-488.2 routine with example programs. The routines are listed alphabetically for easy reference.

- Chapter 3, *NI-488 Function Descriptions*, contains a detailed description of each NI-488 function with example programs. The descriptions are listed alphabetically for easy reference.

- Appendix A, *Multiline Interface Messages*, contains an interface message reference list, which describes the mnemonics and messages that correspond to the interface functions.

- Appendix B, *Applications Monitor*, introduces you to the Applications Monitor, a resident program that is useful in debugging sequences of GPIB calls from within your application.

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

Conventions Used in This Manual

The following conventions are used to distinguish elements of text throughout this manual.

italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept.
`monospace`	Lowercase text in this font denotes text or characters that are to be literally input from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, variables, filenames, and extensions, and for statements and comments taken from program code.
<>	Angle brackets enclose the name of a key on the keyboard–for example, <PageDown>.
<Control>	Key names are capitalized.

In this manual, the term *Software Reference Manual* is used to refer to the *NI-488.2 Software Reference Manual for MS-DOS.*

Abbreviations

The following abbreviations for units of measure are used in this manual:

>	greater than
≥	greater than or equal to
hex	hexadecimal
<	lesser than
msec	millisecond
µsec	microsecond
nsec	nanosecond

Acronyms

The following acronyms are used in this manual:

ANSI	American National Standards Institute
ASCII	American Standard Code for Information Exchange
CIC	Controller-In-Charge
DIO	digital input/output
DMA	direct memory access
EOI	end or identify
EOS	end of string
GPIB	General Purpose Interface (IEEE 488) bus
IEEE 488	Institute of Electrical and Electronic Engineers Standard 488.1-1987, which defines the GPIB
I/O	input/output
PC	personal computer
VAC	volts alternating current

Mnemonics

The following mnemonics are used in this manual:

CIDS	Controller Idle State
DAV	Data Valid
IDY	Identify
NDAC	Not Data Accepted
NRFD	Not Ready For Data
REN	Remote Enable
SRQ	Service Request

Related Documents

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

- *NI-488.2 MS-DOS Software Reference Manual*, part number 320282-01
- ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation*
- ANSI/IEEE Standard 488.2-1987, *IEEE Standard Codes, Formats, Protocals, 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.

Contents

Figure

Tables

Chapter 1
General Information

This chapter lists the files relevant to programming in NKR BASIC, contains programming preparations, discusses how to use the NI-488.2 routine and NI-488 function examples, and summarizes the calls that will be explained at length in Chapter 2 and Chapter 3.

NKR BASIC Files

The NI-488.2 software distribution diskette contains five files relevant to programming in NKR BASIC:

- `DECL.B` is a file containing declarations.
- `NBIB.OBJ` is the NKR BASIC language interface that gives your application program access to the driver.
- `DSAMP.B` is a sample program using device calls.
- `BSAMP.B` is a sample program using board calls.
- `BSAMP488.B` is a sample program using NI-488.2 routines.

Copy the NKR BASIC distribution files to your work area and store the originals in a safe place.

Programming Preparations

Place the following BASIC statement at the beginning of your application program:

```
DECLARE integer ibsta, iberr, ibcnt
```

The following command can be entered from the command line to compile your application program and link it with the language interface object module.

```
b85 -o yourprogramname yourprogramname.b nbib.obj
```

If your application program uses the conversion library, add the option `-lcvt` to the previous command. The NKR BASIC sample programs require this option.

The GPIB status, error, and count information are returned in the variables `ibsta`, `iberr`, and `ibcnt`, respectively.

Testing the Status Word

Testing the value of the status word (`ibsta`) aids in error recovery and diagnostic routines. Notice that the ERR bit is the highest order position of the status word and is therefore the sign bit of the status word. To determine if an error has occurred, test whether the value of `ibsta` is less than zero with the following statement:

```
if (ibsta < 0) then call error
```

where `error` is a user-written error handling routine.

You can also test for particular bits in the status word. Move the value of `ibsta` into a conversion variable denoted by `_i`. The conversion variable is used in the b_AND conversion library function to test the desired bit. The following is an example of testing for the CMPL bits (hex 100):

```
let ibsta_i = ibsta
if b_AND (ibsta_i, 256)<>0 then
   call error
end if
```

Note: Explicit code that tests the status word is not necessary if you are using the applications monitor. For information on the applications monitor refer to Appendix B, *Applications Monitor*.

Count Variable – ibcnt

The count variables are updated after each read, write, or command function with the number of bytes actually transferred by the operation. These variables are also updated by many of the NI-488.2 routines. `ibcnt` is an integer value (16 bits wide).

NKR BASIC NI-488 I/O Calls

The most commonly needed I/O calls are `ibrd` and `ibwrt`. In NKR BASIC, these functions read and write from a character string that can be up to 132 bytes long. The maximum string length can be changed to a value less than 32767. Refer to the *NKR BASIC User's Guide* for methods to increase the string length.

In addition, integer I/O calls (`ibrdi` and `ibwrti`) are provided for users whose data strings are longer than 132 bytes, or who need to perform arithmetic operations on the data and want to avoid the overhead of converting the character bytes of `ibrd` and `ibwrt` into integer format and back again.

`ibrdi` and `ibwrti` are passed data in the form of an integer array, instead of a character string whose maximum length is limited to 132 bytes. Using these functions, you can store more than 132 bytes in a single buffer without having to convert each pair of data bytes to an integer before doing arithmetic operations on the data. Internally, the `ibwrti` function sends each integer to the GPIB in low-byte, high-byte order. The `ibrdi` function reads a series of data bytes from the GPIB and stores them into the integer array in low-byte, high-byte order.

In addition to `ibrdi` and `ibwrti`, the asynchronous functions `ibrdia` and `ibwrtia` are provided to perform asynchronous integer reads and writes.

Using the NI-488.2 Routine and NI-488 Function Examples

Numerous examples are provided with the NI-488 function descriptions in this manual. By including the declaration file, you can pattern your program code after the examples provided.

The routines and functions are listed alphabetically by name in Chapter 2, *NI-488.2 Software Routine Descriptions* and Chapter 3, *NI-488 Function Descriptions*. Tables 1-1 and 1-2 list the NI-488.2 routines and NI-488 functions, respectively, along with a brief descriptions of each routine and function.

Table 1-1. NKR BASIC NI-488.2 Routines

Call Syntax	Description
`AllSpoll (board,addresslist(1), resultlist(1))`	Serial poll all devices
`DevClear (board,address)`	Clear a single device
`DevClearList (board,addresslist(1))`	Clear multiple devices
`EnableLocal (board,addresslist(1))`	Enable operations from the front panel of a device
`EnableRemote (board,addresslist(1))`	Enable remote GPIB programming of devices
`FindLstn (board,addresslist(1), resultlist(1),limit)`	Find all Listeners
`FindRQS (board,addresslist(1), result)`	Determine which device is requesting service
`PassControl (board,address)`	Pass control to another device with Controller capability
`PPoll (board,result)`	Perform a parallel poll
`PPollConfig (board,address, dataline,dataline, sense)`	Configure a device for parallel polls
`PPollUnconfig (board, addresslist(1))`	Unconfigure devices for parallel polls
`RcvRespMsg (board,data$, termination)`	Read data bytes from already addressed device
`ReadStatusByte (board,address, result)`	Serial poll a single device to get its status byte
`Receive (board,address,data$, termination)`	Read data bytes from a GPIB device
`ReceiveSetup (board,address)`	Prepare a particular device to send data bytes and prepare the GPIB board to read them
`ResetSys (board,addresslist(1))`	Initialize a GPIB system on three levels
`Send (board,address,data$,eotmode)`	Send data bytes to a single GPIB device
`SendCmds (board,commands$)`	Send GPIB command bytes

(continues)

Table 1-1. NKR BASIC NI-488.2 Routines (continued)

Call Syntax	Description
`SendDataBytes (board,data$,eotmode)`	Send data bytes to already addressed devices
`SendIFC (board)`	Clear the GPIB interface functions with IFC
`SendList (board,addresslist(1), data$,eotmode)`	Send data bytes to multiple GPIB devices
`SendLLO (board)`	Send the local lockout message to all devices
`SendSetUp (board,addresslist(1))`	Prepare particular devices to receive data bytes
`SetRWLS (board,addresslist)(1)`	Place particular devices in the Remote with Lockout state
`TestSRQ (board,result)`	Determine the current state of the SRQ line
`TestSys (board,addresslist,(1) resultlist(1))`	Cause devices to conduct self-tests
`Trigger (board,address)`	Trigger a single device
`Triggerlist (board,addresslist(1))`	Trigger multiple devices
`WaitSRQ (board,result)`	Wait until a device asserts Service Request

In Table 1-2, the first argument of all function calls except `ibfind` and `ibdev` is the integer variable `ud`, which serves as a unit descriptor. Refer to the *IBFIND* and *IBDEV* function descriptions in Chapter 3, *NI-488 Function Descriptions*, to determine the type of unit descriptor to use.

Note: In function syntax descriptions in this manual, the `ud` argument can also be represented by `bd`, `brd`, or `dev`.

Table 1-2. NKR BASIC NI-488 Functions

Call Syntax	Description
`ibbna (ud,bname$)`	Change access board of device
`ibcac (ud,v)`	Become Active Controller
`ibclr (ud)`	Clear specified device
`ibcmd (ud,cmd$)`	Send commands from string
`ibcmda (ud,cmd$)`	Send commands asynchronously from string
`ibconfig(ud,option,value)`	Configure the handler
`ibdev(bdindex,pad,sad,tmo, eot,eos,ud)`	Open an unused device when device name is unknown
`ibdma (ud,v)`	Enable/disable DMA
`ibeos (ud,v)`	Change/disable EOS mode (write)
`ibeot (ud,v)`	Enable/disable END message
`ibfind (udname$,ud)`	Open device and return unit descriptor
`ibgts (ud,v)`	Go from Active Controller to Standby
`ibist (ud,v)`	Set/clear individual status bit for Parallel Polls
`iblines (ud,clines)`	Get status of GPIB lines
`ibln (ud,pad,sad,listen)`	Check for the presence of a device on the bus
`ibloc (ud)`	Go to Local
`ibonl (ud,v)`	Place device online/offline
`ibpad (ud,v)`	Change Primary Address
`ibpct (ud)`	Pass Control
`ibppc (ud,v)`	Parallel Poll Configure
`ibrd (ud,rd$)`	Read data to string

(continues)

Table 1-2. NKR BASIC NI-488 Functions (continued)

Call Syntax	Description
`ibrda (ud,rd$)`	Read data asynchronously to string
`ibrdf (ud,flname$)`	Read data to file
`ibrdi (ud,iarr(1),cnt)`	Read data to integer array
`ibrdia (ud,iarr(1),cnt)`	Read data asynch to integer array
`ibrpp (ud,ppr)`	Conduct a Parallel Poll
`ibrsc (ud,v)`	Request/release System Control
`ibrsp (ud,spr)`	Return serial poll byte
`ibrsv (ud,v)`	Request service, set/change serial poll byte
`ibsad (ud,v)`	Change Secondary Address
`ibsic (ud)`	Send Interface Clear for 100 µsec
`ibsre (ud,v)`	Set/clear Remote Enable line
`ibstop (ud)`	Abort asynchronous operation
`ibtmo (ud,v)`	Change/disable time limit
`ibtrg (ud)`	Trigger selected device
`ibtrap (mask,mode)`	Configure Applications Monitor
`ibwait (ud,mask)`	Wait for selected event
`ibwrt (ud,wrt$)`	Write data from string
`ibwrta (ud,wrt$)`	Write data asynchronously from string
`ibwrtf (ud,flname$)`	Write data from file
`ibwrti (ud,iarr(1),cnt)`	Write data from integer array
`ibwrtia(ud,iarr(1),cnt)`	Write data asynch from integer array

Dynamic Reconfiguration of Board and Device Characteristics

Some functions can be called during the execution of an application program to dynamically change some of the configured values. These functions are shown in Table 1-3.

Table 1-3. Functions That Alter Default Characteristics

Characteristic	Dynamically Changed by
Primary GPIB address	`ibpad`
Secondary GPIB address	`ibsad`
End-of-string (EOS) byte	`ibeos`
7- or 8-bit compare on EOS	`ibeos`
Set EOI with EOS on Write	`ibeos`
Terminate a Read on EOS	`ibeos`
Set EOI w/last byte of Write	`ibeot`
Change board assignment	`ibbna`
Enable or disable DMA	`ibdma`
Change or disable time limit	`ibtmo`
Request/release system control	`ibrsc`
Set/clear individual status bit	`ibist`
Set/change serial poll status byte	`ibrsv`
Set/clear Remote Enable line	`ibsre`

Chapter 2
NI-488.2 Routine Descriptions

This chapter contains a detailed description of each NI-488.2 routine with example programs. The routines are listed alphabetically for easy reference.

AllSpoll AllSpoll

Purpose: Serial Poll all devices.

Format:
```
CALL AllSpoll(board,addresslist(1),
                    resultlist(1))
```

`board` is a board number. The parameters `addresslist` and `resultlist` are arrays for any size of address intergers, terminated by the value `NOADDR (-1)`. The GPIB devices whose addresses are contained in the `addresslist` array are serial polled, and the responses are stored in the corresponding elements of the `resultlist` array.

If any of the specified devices times out instead of responding to the poll, then the error code EABO is returned in `iberr`, and `ibcnt` contains the index of the timed-out device.

Although the `AllSpoll` routine is general enough to serial poll any number of GPIB devices, the `ReadStatusByte` routine should be used in the case of polling exactly one GPIB device.

Example:

Serial poll two devices connected to board `0` whose GPIB addresses are `8` and `9`.

```
DECLARE integer addresslist (3), resultlist (3)
DECLARE integer board, NOADDR
LET NOADDR = -1
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = NOADDR
CALL AllSpoll(board, addresslist(1), resultlist(1))
```

DevClear DevClear

Purpose: Clear a single device.

Format: `CALL DevClear (board,address)`

`board` is a board number. The GPIB Selected Device Clear (SDC) message is sent to the device at the given address. The parameter `address` contains in its low byte the primary GPIB address of the device to be cleared. The high byte should be 0 if the device has no secondary address. Otherwise, it should contain the desired secondary address. If `address` contains the constant value `NOADDR (-1)`, the universal Device Clear message is sent to all devices on the GPIB.

The `DevClear` routine is used to clear either exactly one GPIB device, or all GPIB devices. To send a single message that clears several particular GPIB devices, use the `DevClearList` routine.

Example:

Clear a digital voltmeter connected to board `0` whose primary GPIB address is `9` and whose secondary GPIB address is `97`.

```
DECLARE integer board, address
LET board = 0
LET address = 9 + 256*97
CALL DevClear (board, address)
```

DevClearList DevClearList

Purpose: Clear multiple devices.

Format: `CALL DevClearList (board,addresslist(1))`

`board` is a board number. The GPIB devices whose addresses are contained in the address array are cleared. The parameter `addresslist` is an array for any size of address integers, terminated by the value `NOADDR (-1)`.

Although the `DevClearList` routine is general enough to clear any number of GPIB devices, the `DevClear` routine should be used in the common case of clearing exactly one GPIB device.

If the array contains only the value `NOADDR`, the universal Device Clear message is sent.

Example:

Clear two devices connected to board `0` whose GPIB addresses are `8` and `9`.

```
DECLARE integer addresslist (3), board, NOADDR
LET NOADDR = -1
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = NOADDR
CALL DevClearList (board, addresslist (1))
```

EnableLocal EnableLocal

Purpose: Enable operations from the front panel of a device.

Format: `CALL EnableLocal (board,addresslist(1))`

`board` is a board number. The GPIB devices whose addresses are contained in the `addresslist` array are placed in local mode by addressing the devices as Listeners and sending the GPIB Go To Local command. The parameter `addresslist` is an array for any size of address integers, terminated by the value `NOADDR (-1)`.

If the array contains only the value `NOADDR`, Remote Enable (REN) becomes unasserted, immediately placing all GPIB devices in local mode.

Example:

Place the devices at GPIB addresses `8` and `9` in local mode.

```
DECLARE integer addresslist (3), board, NOADDR
LET NOADDR = -1
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = NOADDR
CALL EnableLocal(board, addresslist(1))
```

EnableRemote EnableRemote

Purpose: Enable remote GPIB programming of devices.

Format: `CALL EnableRemote (board,addresslist(1))`

`board` is a board number. The GPIB devices whose addresses are contained in the `addresslist` array are placed in remote mode by asserting Remote Enable (REN) and addressing the devices as Listeners. The parameter `addresslist` is an array for any size of address integers, terminated by the value `NOADDR (-1).`

If the array contains only the value `NOADDR`, no addressing is performed, and Remote Enable (REN) becomes asserted.

Example:

Place the devices at GPIB addresses `8` and `9` in remote mode.

```
DECLARE integer addresslist (3), board, NOADDR
LET NOADDR = -1
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = NOADDR
CALL EnableRemote (board, addresslist (1))
```

FindLstn FindLstn

Purpose: Find all Listeners.

Format:
```
CALL FindLstn(board,addresslist(1),
              resultlist(1),limit)
```

`board` is a board number. `addresslist` contains a list of primary GPIB addresses, terminated by the value `NOADDR` (-1). These addresses are tested in turn for the presence of a listening device. If found, the addresses are entered into the `resultlist`. If no listening device is detected at a particular primary address, all the secondary addresses associated with that primary address are tested, and detected Listeners are entered into `resultlist`. The `limit` argument specifies how many entries should be placed into the `resultlist` array. If more Listeners are present on the bus, the list is truncated after `limit` entries have been detected, and the error ETAB will be reported in `iberr`. The variable `ibcnt` will contain the number of addresses placed into `resultlist`.

Because there can be multiple secondary addresses that respond as Listeners for any given primary address, the `resultlist` array should, in general, be larger than the `addresslist` array. In any event, the `resultlist` array (with `limit` being the maximum possible results) must be large enough to accommodate all expected listening devices because no check is made for overflow of the array.

Because most GPIB devices have the ability to listen, this routine is normally used to detect the presence of devices at particular addresses. Once detected, they usually can be interrogated by identification messages to determine what devices they are.

FindLstn (continued) FindLstn

Example:

Determine which one of the devices at addresses `8`, `9`, and `10` are present on the GPIB.

```
DECLARE integer addresslist (4), resultlist (5), board
DECLARE integer limit, NOADDR
!  Because there are three primary GPIB addresses,
!  in the worst case 93 separate GPIB devices could
!  be detected at all the secondary addresses.  In
!  this example, we are assuming that we know that
!  there are at most 5 devices connected to the GPIB.
LET NOADDR = -1
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = 10
LET addresslist (4) = NOADDR
LET limit = 5
CALL FindLstn (board, addresslist (1), resultlist (1),limit)
```

Following this call, the `resultlist` array might contain the following values:

```
resultlist (1)  9
resultlist (2)  10 + 96*256
resultlist (3)  10 + 99*256
```

These results indicate that three GPIB devices were detected. One was found at address `9` with no secondary address, no GPIB devices were detected at primary address `8`, and, at address `10`, two devices with secondary addresses were found. Because only primary GPIB addresses `8`, `9`, and `10` were tested, it is possible that more GPIB devices are connected at other addresses.

FindRQS FindRQS

Purpose: Determine which device is requesting service.

Format: `CALL FindRQS (board,addresslist(1),result)`

`board` is a board number. `addresslist` contains a list of primary GPIB addresses, terminated by the value `NOADDR`. Starting from the beginning of the `addresslist`, the indicated devices are serial polled until one is found asserting SRQ. The status byte for this device is returned in the variable `result`. In addition, the index of the device's address in `addresslist` is returned in the global variable `ibcnt`.

If none of the specified devices is requesting service, the error code ETAB is returned in `iberr`, and `ibcnt` contains the index of the `NOADDR` entry of the list.

If a device times out while responding to its serial poll, the error code EABO is returned in `iberr`, and the index of the timed-out device will appear in `ibcnt`.

Example:

Determine which one of the devices at addresses `8`, `9`, and `10` are requesting service.

```
DECLARE integer addresslist (4), board, result, NOADDR
LET NOADDR = -1
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = 10
LET addresslist (4) = NOADDR
CALL FindRQS (board, addresslist (1), result)
```

Following this call, `result` might contain the value hex 40 (the serial poll response), and `ibcnt` might contain the value 1, indicating that the device at `addresslist (2)` was the first device in the list found to be asserting SRQ.

PassControl PassControl

Purpose: Pass control to another device with Controller capability.

Format: `CALL PassControl (board,address)`

`board` is a board number. The GPIB Device Take Control message is sent to the device at the given address. The parameter `address` contains in its low byte the primary GPIB address of the device to be passed control. The high byte should be 0 if the device has no secondary address. Otherwise, it should contain the desired secondary address.

Example:

Pass control to a Controller connected to board `0` whose primary GPIB address is `9`.

```
DECLARE integer board, address
LET board = 0
LET address = 9
CALL PassControl (board, address)
```

PPoll PPoll

Purpose: Perform a parallel poll.

Format: `CALL PPoll (board,result)`

`board` is a board number. A parallel poll is conducted, and the eight-bit result is stored into `result`. Only the lower eight bits of `result` are affected. The upper byte contains whatever value it did before the call was made.

Each bit of the poll result returns one bit of status information from each device that has been configured for parallel polls. The state of each bit (0 or 1), and the interpretation of these states are based on the latest parallel poll configuration sent to the devices and the individual status of the devices.

Example:

Perform a parallel poll on board `0`.

```
DECLARE integer board, result
LET board = 0
CALL PPoll (board, result)
```

PPollConfig PPollConfig

Purpose: Configure a device for parallel polls.

Format:
```
CALL PPollConfig
(board,address,dataline,sense)
```

`board` is a board number. The GPIB device at `address` is configured for parallel polls according to the `dataline` and `sense` parameters. `dataline` is the data line (1 through 8) on which the device is to respond, and `sense` indicates the condition under which the data line is to be asserted or unasserted. The device is expected to compare this sense value (0 or 1) to its individual status bit, and respond accordingly.

Devices have the option of configuring themselves for parallel polls, in which case they are to ignore attempts by the Controller to configure them. You should determine whether the device is locally or remotely configurable before using `PPollConfig` or `PPollUnconfig`.

Example:

Configure a device connected to board `0` at address `8` so that it responds to parallel polls on data line `5` with sense `0` (assert the line if the individual status is 0, unassert the line if the individual status is 1).

```
DECLARE integer address, board, dataline, sense
LET address = 8
LET board = 0
LET dataline = 5
LET sense = 0
CALL PPollConfig (board, address, dataline, sense)
```

PPollUnconfig PPollUnconfig

Purpose: Unconfigure devices for parallel polls.

Format: `CALL PPollUnconfig (board,addresslist(1))`

`board` is a board number. The GPIB devices whose addresses are contained in the address array are unconfigured for parallel polls; that is, they no longer participate in polls. The parameter `addresslist` is an array of address integers of any size, terminated by the value `NOADDR (-1)`.

If the array contains only the value `NOADDR`, the GPIB Parallel Poll Unconfigure (PPU) message is sent, unconfiguring all devices.

Example:

Unconfigure two devices connected to board 0 whose GPIB addresses are `8` and `9`.

```
DECLARE integer addresslist (3), board, NOADDR
LET NOADDR = -1
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = NOADDR
CALL PPollUnconfig (board, addresslist (1))
```

RcvRespMsg RcvRespMsg

Purpose: Read data bytes from already addressed device.

Format: `CALL RcvRespMsg (board,data$,termination)`

`board` is a board number. The data bytes are read from the GPIB and placed into the pre-allocated string `data`. The amount of data is inferred from the length of the string, which must be pre-allocated to a suitable length. `termination` is a flag used to describe the method of signaling the end of the data. If it is a value between 0 and hex 00FF, the ASCII character with the corresponding hex value is considered the termination character, and the read is stopped when the character is detected. If `termination` is the constant STOPend (defined in the header file `DECL.B`), the read is stopped when EOI is detected.

`RcvRespMsg` assumes that the GPIB Talker and Listeners have already been addressed by a prior call to routines such as `ReceiveSetup`, `Receive`, or `SendCmds`. Thus, it is used specifically to skip the addressing step of GPIB management. The `Receive` routine is normally used to accomplish the entire sequence of addressing followed by the reception of data bytes.

Example:

Receive 100 bytes from an already addressed Talker. The transmission should be terminated when a linefeed character (hex 0A) is detected.

```
DECLARE integer board, termination
LET board = 0
LET data$ = REPEAT$(" ",100)
LET termination = 10
CALL RcvRespMsg (board, data$, termination)
```

ReadStatusByte ReadStatusByte

Purpose: Serial poll a single device to get its status byte.

Format: `CALL ReadStatusByte (board,address,result)`

`board` is a board number. The indicated device is serial polled, and its status byte is placed into the variable `result`, with the status byte zero-extended into the upper byte.

Example:

Serial poll the device at address `8` and return its status byte.

```
DECLARE integer board, address, result
LET board = 0
LET address = 8
CALL ReadStatusByte (board, address, result)
```

Receive Receive

Purpose: Read data bytes from a GPIB device.

Format: `CALL Receive (board,address,data$,termination)`

`board` is a board number. The indicated GPIB device is addressed, and the data bytes are read from that device and placed into the pre-allocated string `data`. `termination` is a value used to describe the method of signaling the end of the data. If it is a value between 0 and hex 00FF, the ASCII character with the corresponding hex value is considered the termination character, and the read is stopped when the character is detected. If `termination` is the constant STOPend (hex 100), the read is stopped when END is detected.

Example:

Receive 100 bytes from the device at address `8`. The transmission should be terminated when END is detected.

```
DECLARE integer board, address, STOPend, termination
LET STOPend = 256
LET board = 0
LET address = 8
LET data$ = REPEAT$ (" ",100)
LET termination = STOPend
CALL Receive (board, address, data$, termination)
```

ReceiveSetup ReceiveSetup

Purpose: Prepare a particular device to send data bytes and prepare the GPIB interface board to read them.

Format: `CALL ReceiveSetup (board,address)`

`board` is a board number. The indicated GPIB device is addressed as a Talker, and the indicated board is addressed as a Listener. Following this routine, it is common to call a routine such as `RcvRespMsg` to actually transfer the data from the Talker.

This routine is useful to initially address devices in preparation for receiving data, followed by multiple calls of `RcvRespMsg` to receive multiple blocks of data, thus eliminating the need to re-address the devices between blocks. Alternatively, the `Receive` routine could be used to send the first data block, followed by `RcvRespMsg` for all the subsequent blocks.

Example:

Prepare a GPIB device at address `8` to send data bytes to board `0`. Then, receive messages of up to 100 bytes from the device, and store it in a string. The message is to be terminated with END.

```
DECLARE integer board, address, STOPend
LET STOPend = 256
LET board = 0
LET address = 8
LET messages$ = REPEAT$(" ",100)
CALL ReceiveSetup (board, address)
LET termination = STOPend
CALL RcvRespMsg (board, messages$, termination)
```

ResetSys ResetSys

Purpose: Initialize a GPIB system on three levels.

Format: `CALL ResetSys (board,addresslist(1))`

`board` is a board number. The GPIB system is initialized on the following three levels:

- Bus initialization: Remote Enable (REN) is asserted, followed by Interface Clear (IFC), causing all devices to become unaddressed and the GPIB interface board (the System Controller) to become the Controller-in-Charge.
- Message exchange initialization: The Device Clear (DCL) message is sent to all connected devices. This ensures that all 488.2 compatible devices can receive the Reset (RST) message that follows.
- Device initialization: `*RST` message is sent to all devices whose addresses are contained in the `addresslist` argument. This causes device-specific functions within each device to be initialized.

Example:

Completely reset a GPIB system containing devices at addresses `8`, `9`, and `10`.

```
DECLARE integer addresslist (4), board, NOADDR
LET board = 0
LET NOADDR = -1
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = 10
LET addresslist (4) = NOADDR
CALL ResetSys (board, addresslist(1))
```

Send Send

Purpose: Send data bytes to a single GPIB device.

Format: `CALL Send (board,address,data$,eotmode)`

`board` is a board number. The indicated GPIB device is addressed as a Listener, the indicated board is addressed as a Talker, and the data bytes contained in `data` are sent. `eotmode` is a flag used to describe the method of signaling the end of the data to the Listener. It should be set to one of the following constants:

- `NLend = 1` Send NL (linefeed) with EOI after the data bytes.
- `DABend = 2` Send EOI with the last data byte in the string.
- `NULLend = 0` Do nothing to mark the end of the transfer.

These constants are defined in the header file `DECL.B.`

Example:

Send an identification query to the GPIB device at address `8`. Terminate the transmission using a linefeed character with END.

```
DECLARE integer board, address, eotmode, NLend
LET NLend = 1
LET board = 0
LET address = 8
LET data$ = “*IDN?”
LET eotmode = NLend
CALL Send (board, address, data$, eotmode)
```

SendCmds SendCmds

Purpose: Send GPIB command bytes.

Format: `CALL SendCmds (board,commands$)`

`board` is a board number. `commands` contains command bytes to be sent onto the GPIB.

`SendCmds` is not normally required for GPIB operation. It is to be used when specialized command sequences, which are not provided for in other routines, must be sent onto the GPIB.

Example:

Controller, at address `0`, simultaneously triggers GPIB devices at addresses `8` and `9`, and immediately places them into local mode.

```
DECLARE integer board
LET board = 0
LET commands$ = chr$(63)&chr$(64)&chr$(40)&
                &chr$(41)&chr$(4)&chr$(1)
CALL SendCmds (board, commands$)
```

SendDataBytes SendDataBytes

Purpose: Send data bytes to already addressed devices.

Format: `CALL SendDataBytes (board,data$,eotmode)`

`board` is a board number. `data` contains data bytes to be sent on to the GPIB. `eotmode` is a flag used to describe the method of signaling the end of the data to the Listeners. It should be set to one of the following constants:

- `NLend = 1` Send NL (linefeed) with EOI after the data bytes.
- `DABend = 2` Send EOI with the last data byte in the string.
- `NULLend = 0` Do nothing to mark the end of the transfer.

These constants are defined in the header file `DECL.B.`

`SendDataBytes` assumes that all GPIB Listeners have already been addressed by a prior call to functions such as `SendSetup`, `Send`, or `SendCmds`. Thus, it is used specifically to skip the addressing step of GPIB management . The `Send` routine is normally used to accomplish the entire sequence of addressing followed by the transmission of data bytes.

Example:

Send an identification query to all addressed Listeners. The transmission should be terminated with a linefeed character with END.

```
DECLARE integer board, NLend, eotmode
LET NLend = 1
LET board = 0
LET data$ = “*IDN?”
LET eotmode = NLend
CALL SendDataBytes (board, data$, eotmode)
```

SendIFC SendIFC

Purpose: Clear the GPIB interface functions with IFC.

Format: `CALL SendIFC (board)`

`board` is a board number. The GPIB Device IFC message is issued, resulting in the interface functions of all connected devices returning to their cleared states.

This function is used as part of GPIB initialization. It forces the GPIB interface board to be Controller of the GPIB, and ensures that the connected devices are all unaddressed and that the interface functions of the devices are in their idle states.

Example:

Clear the interface functions of the devices connected to board `0`.

```
DECLARE integer board
LET board = 0
CALL SendIFC (board)
```

SendList SendList

Purpose: Send data bytes to multiple GPIB devices.

Format:
```
CALL SendList (board,addresslist(1),data$,
                    eotmode)
```

`board` is a board number. `addresslist` contains a list of primary GPIB addresses, terminated by the value `NOADDR (-1)`. The GPIB devices whose addresses are contained in the address array are addressed as Listeners, the indicated board is addressed as a Talker, and the data bytes contained in `data` are sent. `eotmode` is a flag used to describe the method of signaling the end of the data to the Listener. It should be set to one of the following constants:

- `NLend = 1` Send NL (linefeed) with EOI after the data bytes.
- `DABend = 2` Send EOI with the last data byte in the string.
- `NULLend = 0` Do nothing to mark the end of the transfer.

These constants are defined in the header file `DECL.B.`

This routine is similar to `Send`, except that multiple Listeners are able to receive the data with only one transmission.

SendList (continued) SendList

Example:

Send an identification query to the GPIB devices at address 8 and 9. The transmission should be terminated using a linefeed character with EOI.

```
DECLARE integer addresslist (3), board, eotmode
DECLARE integer NOADDR, NLend
LET NLend = 1
LET NOADDR = -1
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = NOADDR
LET data$ = "*IDN?"
LET eotmode = NLend
CALL SendList (board, addresslist(1), data$, eotmode)
```

SendLLO SendLLO

Purpose: Send the Local Lockout message to all devices.

Format: `CALL SendLLO (board)`

`board` is a board number. The GPIB Local Lockout message is sent to all devices, so that the devices cannot independently choose the local or remote states. While Local Lockout is in effect, only the Controller can alter the local or remote state of the devices by sending appropriate GPIB messages.

`SendLLO` is reserved for use in unusual local/remote situations, particularly those in which all devices are to be locked into local programming state. In the typical case of placing devices in Remote Mode With Lockout state, the `SetRWLS` routine should be used.

Example:

Send the Local Lockout message to all devices connected to board `0`.

```
DECLARE integer board
LET board = 0
CALL SendLLO (board)
```

SendSetup SendSetup

Purpose: Prepare particular devices to receive data bytes.

Format: `CALL SendSetup (board,addresslist(1))`

`board` is a board number. The GPIB devices whose addresses are contained in the `addresslist` array are addressed as Listeners, and the indicated board is addressed as a Talker. Following this call, it is common to call a routine such as `SendDataBytes` to actually transfer the data to the Listeners. The parameter `addresslist` is an array for any size of address integers, terminated by the value `NOADDR (-1)`.

This command would be useful to initially address devices in preparation for sending data, followed by multiple calls of `SendDataBytes` to send multiple blocks of data, thus eliminating the need to re-address the devices between blocks. Alternatively, the `Send` routine could be used to send the first data block, followed by `SendDataBytes` for all the subsequent blocks.

Example:

Prepare GPIB devices at addresses `8` and `9` to receive data bytes. Then, send both devices the five messages stored in a string array. EOI is to be sent along with the last byte of the last message.

```
DECLARE integer addresslist (3), board, i
DECLARE integer NULLend, NLend
DECLARE string*10 messages$(5)
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = NOADDR
LET messages$(1) = "Message 0"
LET messages$(2) = "Message 1"
LET messages$(3) = "Message 2"
LET messages$(4) = "Message 3"
LET messages$(5) = "Message 4"
CALL SendSetup (board, addresslist (1))
FOR i = 1 to 4
  CALL SendDataBytes (board, messages$(i), NULLend)
NEXT i
CALL SendDataBytes (board, messages$(5), NLend)
```

SetRWLS SetRWLS

Purpose: Place particular devices in the Remote With Lockout State.

Format: `CALL SetRWLS (board,addresslist(1))`

`board` is a board number. The GPIB devices whose addresses are contained in the `addresslist` array are placed in remote mode by asserting Remote Enable (REN) and addressing the devices as Listeners. In addition, all devices are placed in Lockout State, which prevents them from independently returning to local programming mode without passing through the Controller. The parameter `addresslist` is an array of any size of address integers, terminated by the value `NOADDR (-1)`.

Example:

Place the devices at GPIB addresses `8` and `9` in Remote With Lockout State.

```
DECLARE integer addresslist (3), board, NOADDR
LET NOADDR = -1
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = NOADDR
CALL SetRWLS (board, addresslist (1))
```

TestSRQ TestSRQ

Purpose: Determine the current state of the SRQ line.

Format: `CALL TestSRQ (board,result)`

`board` is a board number. This call places the value 1 in the variable `result` if the GPIB SRQ line is asserted. Otherwise, it places the value of 0 into `result`.

This routine is similar in format to the `WaitSRQ` routine, except that `WaitSRQ` suspends itself waiting for an occurrence of SRQ, whereas `TestSRQ` returns immediately with the current SRQ state.

Example:

Determine the current state of SRQ.

```
DECLARE integer board, result
LET board = 0
CALL TestSRQ (board, result)
IF result = 1 then
    !  SRQ is asserted
ELSE
    !  No SRQ at this time
END IF
```

TestSys TestSys

Purpose: Cause devices to conduct self-tests.

Format:
```
CALL TestSys (board,addresslist(1),
                   resultlist(1))
```

`board` is a board number. The GPIB devices whose addresses are contained in the address array are simultaneously sent a message that instructs them to conduct their self-test procedures. Each device returns an integer code signifying the results of its tests, and these codes are placed into the corresponding elements of the `resultlist` array. The IEEE 488.2 standard specifies that a result code of 0 indicates that the device passed its tests, and any other value indicates that the tests resulted in an error. The variable `ibcnt` contains the number of devices that failed their tests. The parameter `addresslist` is an array of address integers of any size, terminated by the value `NOADDR (-1)`.

Example:

Instruct two devices connected to board `0` whose GPIB addresses are `8` and `9` to perform their self-tests.

```
DECLARE integer addresslist (3), resultlist (2), NOADDR
LET NOADDR = -1
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = NOADDR
CALL TestSys (board, addresslist (1), resultlist(1))
!  If any of the results are non-zero, the
!  corresponding device has failed the test.
```

Trigger Trigger

Purpose: Trigger a single device.

Format: `CALL Trigger (board,address)`

`board` is a board number. The GPIB Group Execute Trigger message is sent to the device at the given address. The parameter `address` contains in its low byte the primary GPIB address of the device to be cleared. The high byte should be 0 if the device has no secondary address. Otherwise, it should contain the desired secondary address. If the address is `NOADDR` (-1), the Group Execute Trigger message is sent with no addressing, thereby triggering all previously addressed Listeners.

The `Trigger` routine is used to trigger exactly one GPIB device. To send a single message that triggers several particular GPIB devices, use the `TriggerList` function.

Example:

Trigger a digital voltmeter connected to board `0` whose primary GPIB address is `9` and whose secondary GPIB address is `97`.

```
DECLARE integer board, address
LET board = 0
LET address = 9 + 256*97
CALL Trigger (board, address)
```

TriggerList TriggerList

Purpose: Trigger multiple devices.

Format: `CALL TriggerList (board,addresslist(1))`

`board` is a board number. The GPIB devices whose addresses are contained in the address array are triggered simultaneously. The parameter `addresslist` is an array of address integers of any size, terminated by the value `NOADDR (-1)`. If the array contains only the value `NOADDR (-1)`, the Group Execute Trigger message is sent without addressing, thereby triggering all previously addressed Listeners.

Although the `TriggerList` routine is general enough to trigger any number of GPIB devices, the `Trigger` function should be used in the common case of triggering exactly one GPIB device.

Example:

Trigger simultaneously two devices connected to board `0` whose GPIB addresses are `8` and `9`.

```
DECLARE integer addresslist (3), board, NOADDR
LET NOADDR = -1
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = NOADDR
CALL TriggerList (board, addresslist (1))
```

WaitSRQ WaitSRQ

Purpose: Wait until a device asserts Service Request.

Format: `CALL WaitSRQ (board,result)`

`board` is a board number. This routine is used to suspend execution of the program until a GPIB device connected to the indicated board asserts the Service Request (SRQ) line. If the SRQ occurs within the timeout period, the variable `result` will be set to the value 1. If no SRQ is detected before the timeout period expires, `result` will be set to 0.

Notice that this call is similar in format to the `TestSRQ` routine, except that `TestSRQ` returns immediately with SRQ status, whereas `WaitSRQ` suspends the program for, at most, the duration of the timeout period waiting for an SRQ to occur.

Example:

Wait for a GPIB device to request service, and then determine which of three devices at addresses `8`, `9`, and `10` requested the service.

```
DECLARE integer addresslist (4), resultlist (3)
DECLARE integer board, result, NOADDR
LET NOADDR = -1
LET board = 0
LET addresslist (1) = 8
LET addresslist (2) = 9
LET addresslist (3) = 10
LET addresslist (4) = NOADDR
CALL WaitSRQ (board, result)
IF result = 1 THEN
    CALL AllSpoll (board, addresslist (1),
                   resultlist(1))
END IF
!  resultlist() now contains the serial poll
responses
!  for the three devices.
```

NI–488.2 Programming Example

You can take full advantage of the IEEE 488.2-1987 standard by using the NI-488.2 routines. These routines are completely compatible with the controller commands and protocols defined in IEEE 488.2.

The NI-488.2 routines are easy to learn and use. Only a few routines are needed for most application programs.

This example illustrates the programming steps that could be used to program a representative IEEE 488.2 instrument from your personal computer using the NI-488.2 routines. The application is written in NKR BASIC. The target instrument is a digital voltmeter (DVM). This instrument is otherwise unspecified (that is, it is not a DVM manufactured by any particular manufacturer). The purpose here is to explain how to use the driver to execute NI-488.2 programming and control sequences and not how to determine those sequences.

Note: For a more detailed description of each step, refer to Chapter 3, *Writing an Advanced Program Using NI-488.2 Routines*, in the getting started manual that you received with your interface board.

1. Load in the definitions of the NI-488.2 routines from a file that is on your distribution diskette.

2. Initialize the IEEE 488 bus and the interface board Controller circuitry so that the IEEE 488 interface for each device is quiescent, and so that the interface board is Controller-In-Charge and is in the Active Controller State (CACS).

3. Find all of the Listeners:

    a. Find all of the instruments attached to the IEEE 488 bus.

    b. Create an array that contains all of the IEEE 488 primary addresses that could possibly be connected to the IEEE 488 bus.

    c. Find out which, if any, device or devices are connected.

4. Send an identification query to each device for identification.

5. Initialize the instrument as follows:

    a. Clear the multimeter.

    b. Send the IEEE 488.2 Reset command to the meter.

6. Instruct the meter to measure volts alternating current (VAC) using auto-ranging (AUTO), to wait for a trigger from the Controller before starting a measurement (TRIGGER 2), and to assert the IEEE 488 Service Request signal line, SRQ, when the measurement has been completed and the meter is ready to send the result (*SRE 16).

7. For each measurement:

    a. Send the `TRIGGER` command to the multimeter. The command `"VAL1?"` instructs the meter to send the next triggered reading to its IEEE 488.2 output buffer.

    b. Wait until the DVM asserts Service Request (SRQ) to indicate that the measurement is ready to be read.

    c. Read the status byte to determine if the measured data is valid or if a fault condition exists. You can find out by checking the message available (MAV) bit, bit 4 in the status byte.

    d. If the data is valid, read 10 bytes from the DVM.

8. End the session.

The NI-488.2 driver supports two interface boards. These boards are referenced by number from your application program. The reference number is zero (0) for the first board and one (1) for the second board. If you installed two boards in your computer, and you do not know which board is 0 and which board is 1, run the configuration utility, `ibconf`. `ibconf` will show you the relationship between the board number and the base address of the board; thereby identifying the board by its base address. Refer to Chapter 2, *Installation and Configuration of NI-488.2 Software* in the *Software Reference Manual* for additional information about running and using `ibconf`.

NKR BASIC Example Program – NI-488.2 Routines

```
!  NKR BASIC Example Program - NI-488.2 Routines

OPTION BASE 0

   DECLARE integer ibsta, iberr, ibcnt    !  GPIB status
                                          !  variables.
   DECLARE integer instruments(32)        !  Array of primary
                                          !  addresses.
   DECLARE integer boardindex             !  Board index.
   DECLARE integer result(32)             !  Array of listen
                                          !  addresses.
   DECLARE integer num_listeners          !  Number of
                                          !  Listeners on GPIB.
   DECLARE integer limit                  !  Maximum number of
                                          !  Listeners on GPIB.
   DECLARE integer mask                   !  Wait mask.
   DECLARE integer k                      !  FOR loop index.
   DECLARE integer v                      !  GPIB function
                                          !  parameter.
   DECLARE integer SRQasserted            !  Set to indicate if
                                          !  SRQ asserted.
   DECLARE integer fluke                  !  Primary address of
                                          !  Fluke 45.
   DECLARE integer statusByte             !  Serial poll
                                          !  response byte.
   DECLARE integer NOADDR                 !  Terminate address
                                          !  list.
   DECLARE integer NLend                  !  Send NL with EOI
                                          !  after transfer.
   DECLARE integer STOPend                !  Stop the read on
                                          !  EOI.

!  Constants used in this application program.

   LET NOADDR = -1
   LET NLend = 1
   LET STOPend = 256
   LET boardindex = 0

   CALL cls_1

!  Your interface board must be the Controller-In-Charge to
find
!  all Listeners on the GPIB.  To accomplish this, the function
!  SendIFC is called.  If the error bit (ERR) is set in ibsta,
!  call gpiberr with an error message.
```

```
   CALL SendIFC(boardindex)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
      LET msg$ = "SendIFC Error"
      CALL gpiberr
      STOP
   END IF

!  Create an array containing all valid GPIB primary
!  addresses.  This array (instruments) will be given to
!  the function FindLstn to find all Listeners.  The
!  constant NOADDR, defined in DECL.B, signifies the
!  end of the array.

   FOR k = 0 to 30
      LET instruments(k) = k
   NEXT k

   LET instruments(31) = NOADDR

!  Print a message to inform the user that the program is
!  searching for all active Listeners.  Find all of the
!  Listeners on the bus.  Store the listen addresses in the
!  array result.  If the error bit (ERR) is set in ibsta,
!  call gpiberr with an error message.

   PRINT "Finding all Listeners on the bus..."
   LET limit = 31

   CALL FindLstn (boardindex, instruments(0), &
   &              result(0), limit)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
      LET msg$ = "FindLstn Error"
      CALL gpiberr
      STOP
   END IF

!  Assign the value of ibcnt to the variable num_listeners.
The
!  GPIB interface board is detected as a Listener on the bus;
!  however, it is not included in the final count of the number
!  of Listeners.  Print the number of Listeners found.

   LET num_listeners = ibcnt - 1
   PRINT "No. of instruments found = ", num_listeners

!  Send the *IDN? command to each device that was found.
!  Your GPIB interface board is at address 0 by default.
!  The board does not respond to *IDN?, so skip it.
!
!  Establish a FOR loop to determine if the Fluke 45 is a
!  Listener on the GPIB.  The variable k serves as a counter
!  for the FOR loop and as the index to the array result.
```

```
   FOR k = 1 to num_listeners

         !  Send the identification query to each listen address
         !  in the array result.  The constant NLend, defined in
         !  DECL.B, instructs the function Send to append a
         !  linefeed character with EOI asserted to the end of
         !  the message.  If the error bit (ERR) is set in
         !  ibsta, call gpiberr with an error message.

            LET cmd$ = "*IDN?"
            CALL Send(boardindex, result(k), cmd$, NLend)
            LET ibsta_i = ibsta
            IF b_AND(ibsta_i,-32768) <> 0 then
              LET msg$ = "Send Error"
              CALL gpiberr
              STOP
            END IF

         !  Read the name identification response returned
         !  from each device.  Store the response in the array
         !  buffer.  The constant STOPend, defined in DECL.B,
         !  instructs the function Receive to terminate the read
         !  when END is detected.  If the error bit (ERR) is set
         !  in ibsta, call gpiberr with an error message.

            LET Reading$ = Repeat$(" ",10)
            CALL Receive(boardindex,result(k),Reading$, &
            &            STOPend)
            LET ibsta_i = ibsta
            IF b_AND(ibsta_i,-32768) <> 0 then
              LET msg$ = "Receive Error"
              CALL gpiberr
              STOP
            END IF

         !  The low byte of the listen address is the primary
         !  address.  Assign the variable pad the primary
         !  address of the device.

            LET num_i = result(k)
            LET pad_i = b_AND(num_i,255)

         !  Print the measurement received from the Fluke 45.

            LET rd$ = Reading$(1:ibcnt-1)
            PRINT "The instrument at address ";pad_i; " &
            &     is: ";rd$

         !  Determine if the name identification is the Fluke
45.
         !  If it is the Fluke 45, assign pad to fluke, print
         !  message that the Fluke 5 has been found, call the
         !  function Found, and terminate the FOR loop.

            IF left$(Reading$, 9)="FLUKE, 45" & then GOSUB 2000
   NEXT k
```

```
   PRINT "Did not find the Fluke!"
   GOSUB 4000


!  Device found.

2000
   PRINT "**** We found the Fluke 45 ****"
   LET fluke = result(k)

!  Reset the Fluke 45 using the functions DevClear and
!  Send.
!
!  DevClear will send the GPIB Selected Device Clear (SDC)
!  command message to the Fluke 45.  If the error bit (ERR)
!  is set in ibsta, call gpiberr with an error message.

   CALL DevClear (boardindex, fluke)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "DevClear Error"
     CALL gpiberr
     STOP
   END IF

!  Use the function Send to send the IEEE 488.2 reset command
!  (*RST) to the Fluke 45.  The constant NLend, defined
!  in DECL.B, instructs the function Send to append a linefeed
!  character with EOI asserted to the end of the message.
!  If the error bit (ERR) is set in ibsta, call gpiberr
!  with an error message.

   LET cmd$ = "*RST"
   CALL Send(boardindex, fluke, cmd$, NLend)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "Send *RST Error"
     CALL gpiberr
     STOP
   END IF

!  Use the function Send to send device configuration commands
!  to the Fluke 45.  Instruct the Fluke 45 to measure volts
!  alternating current (VAC) using auto-ranging (AUTO),
!  to wait for a trigger from the GPIB interface board
!  (TRIGGER 2), and to assert the IEEE 488 Service Request
!  line, SRQ, when the measurement has been completed and
!  the Fluke 45 is ready to send the result (*SRE 16).  If
!  the error bit (ERR) is set in ibsta, call gpiberr with an
!  error message.
```

```
   LET cmd$ = "VAC; AUTO; TRIGGER 2; *SRE 16"
   CALL Send(boardindex, fluke, cmd$, NLend)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "Send setup Error"
     CALL gpiberr
     STOP
   END IF

!  Initialize the accumulator of the ten measurements to zero.

   LET sum = 0

!  Establish a FOR loop to read the ten measurements.  The
!  variable m serves as the counter of the FOR Loop.

   FOR m = 1 to 10

        !  Trigger the Fluke 45 by sending the trigger command
        !  (*TRG) and request a measurement by sending the
        !  command "VAL1?".  If the error bit (ERR) is set in
        !  ibsta, call gpiberr with an error message.

           LET cmd$ = "*TRG; VAL1?"
           CALL Send(boardindex, fluke, cmd$, NLend)
           LET ibsta_i = ibsta
           IF b_AND(ibsta_i,-32768) <> 0 then
             LET msg$ = "Send trigger Error"
             CALL gpiberr
             STOP
           END IF

        !  Wait for the Fluke 45 to assert SRQ, meaning it is
        !  ready to send a measurement.  If SRQ is not
        !  asserted within the timeout period, call gpiberr
        !  with an error message.  The timeout period by
        !  default is 10 seconds.

           CALL WaitSRQ(boardindex, SRQasserted)
           IF SRQasserted = 0 then
             LET msg$ = "SRQ is not asserted.  The Fluke is &
             &           not ready."
             CALL gpiberr
             STOP
           END IF
```

```
!  Read the serial poll status byte of the Fluke 45.
!  If the error bit (ERR) is set in ibsta, call gpiberr
!  with an error message.

   CALL ReadStatusByte(boardindex, fluke, &
   &    statusByte)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "ReadStatusByte Error"
     CALL gpiberr
     STOP
   END IF

!  Check if the Message Available Bit (bit 4) of the
!  return status byte is set.  If this bit is not set,
!  print the status byte and call gpiberr with an
!  error message.

   LET mask = 16
   LET status_i = statusByte
   IF b_AND(status_i,mask) <> 16 then
     LET msg$ = "Improper Status Byte"
     CALL gpiberr
     PRINT "Status Byte = "; statusByte
     STOP
   END IF

!  Read the Fluke 45 measurement.  Store the
!  measurement in the array Reading.  The constant
!  STOPend, defined in DECL.B, instructs the function
!  Receive to terminate the read when END is detected.
!  If the error bit (ERR) is set in ibsta, call gpiberr
!  with an error message.

   CALL Receive (boardindex, fluke, Reading$, & STOPend
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "Receive Error"
     CALL gpiberr
     STOP
   END IF

!  Print the measurement received from the Fluke 45.

   LET rd$ = Reading$(1:ibcnt-1)
   PRINT "Reading : ";rd$
   PRINT
```

```
        !  Convert the measurement to its numeric value.
        !  If there is an error during the conversion,
terminate
        !  this program.  If an error does not occur during
this
        !  conversion, add the value to the accumulator.

           LET sum = sum + val(Rd$)
   NEXT m

   PRINT "The average of the 10 readings is ", sum/10

!  Call the ibonl function to disable the hardware and
!  software.

4000

   LET v = 0
   CALL ibonl (boardindex,v)


! ============================================================
!                        Procedure gpiberr
!  The gpiberr procedure notifies you that an NI-488.2 routine
!  failed by printing an error message.  The status variable
!  ibsta prints in hexadecimal along with the mnemonic
!  meaning of the bit position.  The status variable iberr
!  prints in decimal along with the mnemonic meaning of the
!  decimal value.  The status variable ibcnt prints in decimal.
!
!  The NI-488 function ibonl disables the hardware and
!  software.
! ============================================================
SUB gpiberr
   PRINT msg$

   PRINT "ibsta = &H"; hex$(ibsta_i)

   IF b_AND(ibsta_i, -32768)    <>  0 then PRINT " ERR"
   IF b_AND(ibsta_i, 16384)     <>  0 then PRINT " TIMO"
   IF b_AND(ibsta_i, 8192)      <>  0 then PRINT " END"
   IF b_AND(ibsta_i, 4096)      <>  0 then PRINT " SRQI"
   IF b_AND(ibsta_i, 2048)      <>  0 then PRINT " RQS"
   IF b_AND(ibsta_i, 256)       <>  0 then PRINT " CMPL"
   IF b_AND(ibsta_i, 128)       <>  0 then PRINT " LOK"
   IF b_AND(ibsta_i, 64)        <>  0 then PRINT " REM"
   IF b_AND(ibsta_i, 32)        <>  0 then PRINT " CIC"
   IF b_AND(ibsta_i, 16)        <>  0 then PRINT " ATN"
   IF b_AND(ibsta_i, 8)         <>  0 then PRINT " TACS"
   IF b_AND(ibsta_i, 4)         <>  0 then PRINT " LACS"
   IF b_AND(ibsta_i, 2)         <>  0 then PRINT " DTAS"
   IF b_AND(ibsta_i, 1)         <>  0 then PRINT " DCAS"
   PRINT
```

```
   PRINT "iberr = ", iberr
   IF iberr =  0 then PRINT " EDVR <DOS Error>"
   IF iberr =  1 then PRINT " ECIC <Not CIC>"
   IF iberr =  2 then PRINT " ENOL <No Listener>"
   IF iberr =  3 then PRINT " EADR <Address error>"
   IF iberr =  4 then PRINT " EARG <Invalid argument>"
   IF iberr =  5 then PRINT " ESAC <Not Sys Ctrlr>"
   IF iberr =  6 then PRINT " EABO <Op. aborted>"
   IF iberr =  7 then PRINT " ENEB <No GPIB board>"
   IF iberr = 10 then PRINT " EOIP <Async I/O in prg>"
   IF iberr = 11 then PRINT " ECAP <No capability>"
   IF iberr = 12 then PRINT " EFSO <File sys. error>"
   IF iberr = 14 then PRINT " EBUS <Command error>"
   IF iberr = 15 then PRINT " ESTB <Status byte lost>"
   IF iberr = 16 then PRINT " ESRQ <SRQ stuck on>"
   IF iberr = 20 then PRINT " ETAB <Table Overflow>"
   PRINT

   PRINT "ibcnt = ", ibcnt

!  Call the ibonl function to disable the hardware
!  and software.

   LET v = 0
   CALL ibonl (boardindex,v)

END SUB

END
```

Chapter 3
NI-488 Function Descriptions

This chapter contains a detailed description of each NI-488 function with example programs. The descriptions are listed alphabetically for easy reference.

In the following function syntax descriptions, the `ud` argument can also be represented by `bd`, `brd`, or `dev`.

IBBNA IBBNA

Purpose: Change access board of device.

Format: `CALL ibbna (ud,bname$)`

`ud` is a device. `bname` is the new access board to be used in all device calls to that device. `ibbna` is needed only to alter the board assignment from its configuration setting.

The assigned board is used in all subsequent device functions used with that device until `ibbna` is called again, `ibonl` or `ibfind` is called, or the system is restarted.

Refer also to Table 1-2.

Device Function Example:

Associate the device `DVM` with the interface board `"GPIB0"`.

```
DECLARE integer dvm
LET udname$ = "DVM"
CALL ibfind (udname$,dvm)
LET bname$ = "GPIB0"
CALL ibbna (dvm, bname$)
```

IBCAC IBCAC

Purpose: Become Active Controller.

Format: `CALL ibcac (ud,v)`

`ud` is an interface board. If `v` is non-zero, the GPIB board takes control synchronously with respect to data transfer operations; otherwise, the GPIB board takes control immediately (asynchronously).

To take control synchronously, the GPIB board asserts the ATN signal without corrupting data being transferred. If a data handshake is in progress, the take control action is postponed until the handshake is complete; if a handshake is not in progress, the take control action is done immediately. Synchronous take control is not guaranteed if an `ibrd` or `ibwrt` operation completed with a timeout error.

Asynchronous take control should be used in situations where it appears to be impossible to gain control synchronously (for example, after a timeout error).

It is generally not necessary to use the `ibcac` function in most applications. Functions such as `ibcmd` and `ibrpp`, which require that the GPIB board take control, do so automatically.

The ECIC error results if the GPIB board is not Controller-In-Charge (CIC).

IBCAC (continued) IBCAC

Board Function Examples:

1. Take control immediately without regard to any data handshake in progress.

```
DECLARE integer v, brd0
LET v = 0
CALL ibcac (brd0,v)
!  ibsta should show that the interface board is now
!  CAC, that is, CIC with ATN asserted.
```

2. Take control synchronously and assert ATN following a read operation.

```
DECLARE integer brd0
LET board$ = "GPIB0"
CALL ibfind (brd0, board$)
CALL ibrd (brd0, rd$)
LET v = 1
CALL ibcac (brd0, v)
```

IBCLR IBCLR

Purpose: Clear specified device.

Format: `CALL ibclr (ud)`

`ud` is a device.

The `ibclr` function clears the internal or device functions of a specified device.

`ibclr` calls the board function `ibcmd` to send the following commands using the designated access board:

- Talk address of access board
- Unlisten (UNL)
- Listen address of the device
- Secondary address of the device, if applicable
- Selected Device Clear (SDC)

Other command bytes may be sent as necessary.

Refer to *IBCMD* for additional information.

Device Function Example:

Clear the device `vmtr`.

```
DECLARE integer vmtr
LET dev$ = "DEV3"           !  open the voltmeter
CALL ibfind (dev$, vmtr)
!  Clear the voltmeter
CALL ibclr (vmtr)
```

IBCMD IBCMD

Purpose: Send commands from string.

Format: `CALL ibcmd (ud,cmd$)`

`ud` is an interface board. `cmd` contains the commands to be sent over the GPIB.

The `ibcmd` function is used to transmit interface messages (commands) over the GPIB. These commands are listed in Appendix A, *Multiline Interface Messages*. The `ibcmd` function is also used to pass GPIB control to another device. This function is *not* used to transmit programming instructions to devices. These instructions are transmitted with the `ibrd` and `ibwrt` functions.

The `ibcmd` operation terminates on any of the following events:

- All commands are successfully transferred.
- An error is detected.
- The time limit is exceeded.
- A Take Control (TCT) command is sent.
- An Interface Clear (IFC) message is received from the System Controller.

The transfer count may be less than the requested count on any of the previous terminating events but the first.

An ECIC error results if the GPIB board is not CIC. If it is not Active Controller, the GPIB board takes control and asserts ATN prior to sending the command bytes. The GPIB board remains Active Controller afterward.

In the examples that follow, GPIB commands and addresses are coded as printable ASCII characters. If values correspond to printable ASCII characters, it is simplest to use the ASCII characters to specify the values. Refer to Appendix A, *Multiline Interface Messages* for the ASCII characters corresponding to a numeric value.

IBCMD (continued) IBCMD

Board Function Examples:

1. Unaddress all Listeners with the Unlisten (UNL or ASCII ?) command and address a Talker at hex 46 (ASCII F) and a Listener at hex 31 (ASCII 1).

```
DECLARE integer brd0
LET cmd$ = "?F1"                ! UNL TAD1 LAD2
CALL ibcmd (brd0, cmd$)
```

2. Same as Example 1, except the Listener has a secondary address of hex 6E (ASCII n).

```
DECLARE integer brd0
LET cmd$ = "?F1n"               ! UNL TAD1 LAD2 SAD2
CALL ibcmd (brd0, cmd$)
```

3. Clear all GPIB devices (that is, reset internal functions) with the Device Clear (DCL or hex 14) command.

```
DECLARE integer brd0
LET cmd$ = chr$ (20)            ! DCL
CALL ibcmd (brd0, cmd$)
```

4. Clear two devices with listen addresses of hex 21 (ASCII !) and hex 28 (ASCII ([left parenthesis]) with the Selected Device Clear (SDC or hex 04) command.

```
LET cmd$ = "!(" & chr$(04)      ! LAD1 LAD2
CALL ibcmd (brd0, cmd$)         ! SDC
```

5. Trigger any devices previously addressed to listen with the Group Execute Trigger (GET or hex 08) command.

```
DECLARE integer brd0
LET cmd$ = chr$(08)             ! GET
CALL ibcmd (brd0, cmd$)
```

IBCMD (continued) IBCMD

6. Serial poll a device at talk address hex 52 (ASCII R) using the Serial Poll Enable (SPE or hex 18) and Serial Poll Disable (SPD or hex 19) commands (the GPIB board listen address is hex 20 or ASCII <space>).

```
DECLARE integer brd0
LET cmd$ = "R " + chr$(24)      !  UNL TAD MLA SPE
CALL ibcmd (brd0, cmd$)
LET rd$ = repeat$(" ",1)
CALL ibcmd (brd0, rd$)
!  After checking the status byte in rd$, disable
!  this device and unaddress it with the Untalk
!  (UNT or ASCII _) command before polling the next
one.
LET cmd$ = chr$(25) & "_"       !  SPD UNT
CALL ibcmd (brd0, cmd$)
```

IBCMDA IBCMDA

Purpose: Send commands asynchronously from string.

Format: `CALL ibcmda (ud,cmd$)`

`ud` is an interface board. `cmd` contains the commands to be sent over the GPIB.

The `ibcmda` function is used to transmit interface messages (commands) over the GPIB. These commands are listed in Appendix A, *Multiline Interface Messages*. The `ibcmda` function can also be used to pass GPIB control to another device. This function is *not* used to transmit programming instructions to devices. These instructions and other device-dependent information are transmitted with the `ibrd` and `ibwrt` functions.

`ibcmda` is used in place of `ibcmd` if the application program must perform other functions while processing the GPIB command. `ibcmda` returns immediately after starting the I/O operation.

The three asynchronous I/O calls (`ibcmda` , `ibrda`, and `ibwrta`) are designed to allow an application to perform other functions (non-GPIB functions) while processing the I/O. Once the asynchronous I/O call has been initiated, further GPIB calls involving the device or access board are not allowed until the I/O has completed and the GPIB driver and the application have been resynchronized.

Resynchronization can be accomplished by using one of the following three functions:

Note: Resynchronization is only successful if the `ibsta` returned contains CMPL.

- `ibwait` (mask contains CMPL) - The driver and application are synchronized.
- `ibstop` - The asynchronous I/O is canceled, and the driver and application are synchronized.
- `ibonl` - The asynchronous I/O is canceled, the interface has been reset, and the driver and application are synchronized.

IBCMDA (continued) IBCMDA

The only other GPIB call that is allowed during asynchronous I/O is the `ibwait` function (mask is arbitrary). Any other GPIB call involving the device or access board returns the EOIP error.

An ECIC error results if the GPIB board is not CIC. If it is not Active Controller, the GPIB board takes control and asserts ATN prior to sending the command bytes. It remains Active Controller afterward. The ENOL error will be returned if there are no other devices on the IEEE 488 bus.

Board Function Example:

Address several devices for a broadcast message to follow while testing for a high priority event to occur.

```
DECLARE integer brd0, mask
!  The interface board BRD0 at talk address hex 40
!  (ASCII @), addresses nine Listeners at addresses
!  hex 31-hex 39 (ASCII 1-9) to receive the
!  broadcast message.
LET board$ = "GPIB0"
CALL ibfind (board$, brd0)
LET cmd$ = "?@1234567789"      !  UNL MTA
                               !  LAD1...LAD9
CALL ibcmda (brd0, cmd$)
LET ibsta_i = 0
LET mask = 0                   !  Set mask to return
                               !  immediately.
Do while b_AND (ibsta_i, 256) <> 256
   CALL eventest               !  HIGH PRIORITY
                               !  ROUTINE
   CALL ibwait (brd0, mask)
   LET ibsta_i = ibsta
   IF ibsta <0 then GOSUB ERROR
WEND
PRINT "Asynchronous commands sent!"
LET mask = 16640
CALL ibwait (brd0, mask)
PRINT "Asynchronous transfer properly terminated."
```

IBCONFIG IBCONFIG

Purpose: Change the driver configuration parameters.

Format: `CALL ibconfig (ud,option,value)`

`ud` is a GPIB interface board or a device. `option` is used to select the configurable item in the driver. The configurable item is set to the contents of `value`. The previous contents of the configurable item is returned in `iberr`. If `ud` is a GPIB interface board descriptor, `option` takes on the values shown in Table 3-1. If `ud` is a device descriptor, `option` has the values shown in Table 3-2.

Table 3-1. Board Configuration Options

Option	Description
1	Primary Address. `value` is the new primary address of the GPIB interface board (0–30). See *IBPAD* and Appendix A.
2	Secondary Address. `value` is the new secondary address of the board (0, 96–126). See *IBSAD* and Appendix A.
3	Timeout Value. `value` is the new timeout `value` of the board (0–17). See *IBTMO*.
4	Enable/disable END message on write operations. `value` is the new EOT mode (0 = no END, non-zero = send END with last byte). See *IBEOT*.
5	Parallel Poll Configure. `value` is the parallel poll configure byte (0, 96–126). See *IBPPC*.
7	Enable/disable Automatic Serial Polling. If `value` is zero (0), Autopolling is disabled. If `value` is non-zero, Autopolling is enabled.
8	Use/do not use the NI-488 CIC protocol. If `value` is zero (0), do not use the CIC protocol. If `value` is non-zero, use the CIC protocol. See the *Device Functions* section in Chapter 5 of the *Software Reference Manual.*

(continues)

IBCONFIG (continued) IBCONFIG

Table 3-1 Board Configuration Options (continued)

Option	Description
9	Enable/disable hardware interrupts. If `value` is zero (0), disable GPIB interface board interrupts. If `value` is non-zero, enable GPIB interface board interrupts. See description of the IBCONF utility program in Chapter 2 of the *Software Reference Manual*.
10	Request or release System Control. If `value` is zero (0), functions requiring System Controller capability are not allowed. If `value` is non-zero, functions requiring System Controller capability are allowed. See *IBRSC*.
11	Assert/unassert REN. If `value` is non-zero, the IEEE 488 Remote Enable (REN) signal is asserted. If `value` is zero (0), REN is unasserted. See *IBSRE*.
12	Terminate read when End-Of-String (EOS) character is detected. If `value` is non-zero, read functions are terminated when the EOS character is detected in the data stream. If `value` is zero, EOS detection is disabled. See *IBEOS*.
13	Assert EOI when sending EOS character. If `value` is zero (0), do not send EOI with EOS. If `value` is non-zero, send EOI with EOS. See *IBEOS*.
14	Use 7- / 8-bit EOS comparison. If `value` is zero, use low-order 7 bits of EOS character for comparison. If `value` is non-zero, use 8 bits. See *IBEOS*.
15	End-Of-String (EOS) character. `value` is the new EOS character of the board (8 bits). See *IBEOS*.

(continues)

IBCONFIG (continued) IBCONFIG

Table 3-1 Board Configuration Options (continued)

Option	Description
16	Parallel Poll remote/local configuration. If `value` is zero, the GPIB interface board uses IEEE 488 Parallel Poll (PP) interface function subset PP1 (remote configuration by external Controller). If `value` is non-zero, the board uses PP subset PP2 (local configuration from your application program: `value` is used as the local poll enable [lpe] message). See *IBPPC*.
17	IEEE 488 bus handshake timing. If `value` is one (1), normal timing is used for the IEEE 488 Source Handshake T1 delay (≥ 2 µsec). If `value` is two (2), high-speed timing is used for T1 (≥ 500 nsec). If `value` is three (3), very high-speed timing is used (≥ 350 nsec).
18	Enable/disable direct memory access (DMA) transfers. If `value` is zero (0), disable GPIB interface board DMA transfers. If `value` is non-zero, enable GPIB interface board DMA transfers. See the description of the IBCONF utility program in Chapter 2 of the *Software Reference Manual*.
19	Byte swapping on `ibrd`. If `value` is one (1), pairs of bytes read off the bus are swapped before storing them in the ibrd buffer. The transfer count must be even or ECAP will be returned. In this case, the last two bytes of the buffer will be invalid. If ECAP is returned and your buffer begins on an odd address, start the buffer on an even address. If `value` is zero (0), byte swapping on `ibrd` is disabled.
20	Byte swapping on `ibwrt`. If `value` is one (1), pairs of bytes are swapped before they are written from the user's buffer to the bus. The transfer count must be even or ECAP will be returned. In some cases, the address of the buffer must be even. If ECAP is returned and your buffer begins on an odd address, start the buffer on an even address. If `value` is zero (0), byte swapping on `ibwrt` is disabled.

IBCONFIG (continued) IBCONFIG

Table 3-2. Device Configuration Options

Option	Description
1	Primary Address. `value` is the new primary address of the device (0–30). See *IBPAD* and Appendix A.
2	Secondary Address. `value` is the new secondary address of the device (0, 96–126). See *IBSAD* and Appendix A.
3	Timeout Value. `value` is the new timeout value of the device (0–17). See *IBTMO*.
4	Enable/disable END message on write operations. `value` is the new EOT mode (0 = no END, non-zero = send END with last byte). See *IBEOT*.
6	Repeat Addressing. If value is zero (0), disable repeat addressing. If `value` is non-zero (1), enable repeat addressing. See the description of the IBCONF utility in Chapter 2 of the *Software Reference Manual*.
12	Terminate read when End-Of-String (EOS) character is detected from this device. If `value` is non-zero, read functions are terminated when the EOS character is detected in the data stream received from the device. If `value` is zero, EOS detection is disabled. See *IBEOS*.
13	Assert EOI when sending EOS character to this device. If value is zero (0), do not send EOI with EOS. If `value` is non-zero, send EOI with EOS. See *IBEOS*.
14	Use 7- / 8-bit EOS comparison. If `value` is zero, use low-order 7 bits of EOS character for comparison. If `value` is non-zero, use 8 bits. See *IBEOS*.
15	End-Of-String (EOS) character. `value` is the new EOS character (8 bits) to use with this device. See *IBEOS*.

(continues)

IBCONFIG (continued) IBCONFIG

Table 3-2. Device Configuration Options (continued)

Option	Description
19	Byte swapping on `ibrd`. If `value` is one (1), pairs of bytes read off the bus are swapped before storing them in the `ibrd` buffer. The transfer count must be even or ECAP will be returned. In this case, the last two bytes of the buffer will be invalid. If ECAP is returned and your buffer begins on an odd address, start the buffer on an even address. If `value` is zero (0), byte swapping on `ibrd` is disabled.
20	Byte swapping on `ibwrt`. If `value` is one (1), pairs of bytes are swapped before they are written from the user's buffer to the bus. The transfer count must be even or ECAP will be returned. In some cases, the address of the buffer must be even. If ECAP is returned and your buffer begins on an odd address, start the buffer on an even address. If `value` is zero (0), byte swapping on `ibwrt` is disabled.

IBCONFIG (continued) IBCONFIG

Device Function Example

Set up various configurable parameters in preparation for a device read.

```
DECLARE dev1, option, value
LET dev$ = "dev1"
CALL ibfind (dev$, dev1)
!  Enable repeat addressing
LET option = 6
LET value = 1
CALL ibconfig (dev1, option, value)
!  Set linefeed as the EOS character
LET option = 15
LET value = 10
CALL ibconfig (dev1, option, value)
!  Use 7-bit comparison for EOS character
LET option = 14
LET value = 0
CALL ibconfig (dev1,option,value)
!  Terminate reads on EOS
LET option = 12
LET value = 1
CALL ibconfig (dev1,option,value)
```

IBCONFIG (continued) IBCONFIG

Board Function Example:

1. Set up various configurable parameters in preparation for a board read.

```
DECLARE integer gpib0, option, value
LET gpib$ = "gpib0"
CALL ibfind (gpib$, gpib0)
!  Enable DMA transfers
LET option = 18
LET value = 1
CALL ibconfig (gpib0, option, value)
!  Turn off Autopolling
LET option = 7
LET value = 0
CALL ibconfig (gpib0, option, value)
!  Turn on interrupts
LET option = 9 : value = 1
CALL ibconfig (gpib0, option, value)
```

IBCONFIG (continued) IBCONFIG

2. Enable automatic byte swapping of binary integer data.

```
      DECLARE integer array (500), option, value, count

!  Read in unswapped header data

      LET header$ = repeat$(" ",10)
      CALL ibrd (ud, header$)

!  Arrange for byte swapping

      LET option = 19
      LET value = 1
      CALL (ud, option, value)

!  Read 1,000 bytes with automatic swapping

      LET count = 1000
      CALL ibrdi (ud, array (1), count)

!  Disable swapping for subsequent reads

      LET value = 0
      CALL ibconfig (ud, option, value)
```

IBDEV IBDEV

Purpose: Open and initialize an unused device when device name is unknown.

Format: `CALL ibdev (bdindex,pad,sad,tmo,eot,eos,ud)`

`bdindex` is an index from 0 to [(number of boards) - 1] of the access board with which the device descriptor must be associated. The arguments `pad`, `sad`, `tmo`, `eot`, and `eos` dynamically set the software configuration for the NI-488 I/O functions. These arguments configure the primary address, secondary address, I/O timeout, asserting EOI on last byte of date sourced, and the End-Of-String mode and byte, respectively. (Refer to *IBPAD*, *IBSAD*, *IBTMO*, *IBEOT*, and *IBEOS* for more information on each argument.) The device descriptor is returned in the variable `ud`.

The `ibdev` command selects an available device, opens it, and initializes it. You can use this function in place of `ibfind`.

`ibdev` returns a device descriptor of the first unopened user configured device that it finds. For this reason, it is very important to use `ibdev` *only after* all of your `ibfind` calls have been made. This is the only way to ensure that `ibdev` does not use a device that you plan to use via an `ibfind` call. The `ibdev` function performs the equivalent `ibonl` to open the device.

Note: The device descriptor of the NI-488.2 driver can remain open across invocations of an application, so be sure to return the device descriptor to the pool of available devices by calling `ibonl` with `v`=0 when you are finished using the device. If you do not, that device will not be available for the next `ibdev` call.

If the `ibdev` call fails, a negative number is returned in place of the device descriptor. There are two distinct errors that can occur with the `ibdev` call:

- If no device is available or the specified board index refers to a non-existent board, `ibdev` returns the EDVR or ENEB error.
- If one of the last five parameters is an illegal value, `ibdev` returns with a good board descriptor and the EARG error.

IBDEV (continued) IBDEV

Device Function Example:

`ibdev` opens an available device and assigns it to access `gpib0` (board = 0) with a primary address of 6 (`pad` = 6), a secondary address of hex 67 (`sad` = 113), a timeout of 10 msec (`tmo` = 7), the END message enabled (`eot` = 1) and the EOS mode disabled (`eos` = 0).

```
DECLARE integer bdindex, pad,sad,tmo
DECLARE integer eot,eos,ud,EDVR,EARG
LET EDVR = 0
LET EARG = 4
!  Get a device descriptor associated with board 0.
LET bdindex = 0
LET pad = 6
LET sad = 113
LET tmo = 7
LET eot = 1
LET eos = 0
CALL ibdev (bdindex,pad,sad,tmo,eot,eos,ud)
IF ud < 0                    !  Handle GPIB error here
     IF iberr = EDVR then
!  Bad bdindex or no devices available else if
!  iberr = EARG then the call succeeded, but at
!  least one of pad,sad,tmo,eos,eot is incorrect.
     END IF
END IF
```

IBDMA IBDMA

Purpose: Enable or disable DMA.

Format: `CALL ibdma (ud,v)`

`ud` is an interface board. If `v` is non-zero, DMA transfers between the GPIB board and memory are used for read and write operations. If `v` is zero, programmed I/O is used.

If you enabled DMA at configuration time, this function can be used to switch between programmed I/O and DMA using the selected channel. If you disabled DMA at configuration time or if your computer does not have DMA capability, calling this function with `v` equal to a non-zero value results in an ECAP error.

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

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

Also refer to Table 1-2.

Board Function Examples:

1. Enable DMA transfers using the previously configured channel.

```
DECLARE integer brd0, v
LET v = 1             !  Any non-zero value will do.
CALL ibdma (brd0,v)
```

2. Disable DMAs and use programmed I/O exclusively.

```
LET v = 0
CALL ibdma (brd0, v)
```

IBEOS IBEOS

Purpose: Change or disable End-Of-String termination mode.

Format: `CALL ibeos (ud,v)`

`ud` is a device or an interface board. `v` specifies the EOS character and the data transfer termination method according to Table 3-3. `ibeos` is needed only to alter the value from its configuration setting.

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

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

Table 3-3. Data Transfer Termination Method

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

Methods A and C determine how read operations terminate. If Method A alone is chosen, reads terminate when the low seven bits of the byte that is read match the low seven bits of the EOS character. If Methods A and C are chosen, a full 8-bit comparison is used.

IBEOS (continued) IBEOS

Methods B and C together determine when write operations send the END message. If Method B alone is chosen, the END message is sent automatically with the EOS byte when the low seven bits of that byte match the low seven bits of the EOS character. If Methods B and C are chosen, a full 8-bit comparison is used.

Note: Defining an EOS byte for a device or board does not cause the driver to automatically send that byte when performing writes. Your application program must include the EOS byte in the data string it defines.

Device IBEOS Function

If ud is a device, the options coded in v are used for all device reads and writes in which that device is specified.

Board IBEOS Function

If ud is a board, the options coded in v become associated with all board reads and writes.

Refer also to *IBEOT* and Table 1-2.

Device Function Example:

Send END when the linefeed character (hex 0A) is written to the device dvm.

```
DECLARE integer v, EOSV
LET EOSV = 10                    !  EOS info for
                                 !  ibeos.


LET v = EOSV + 2048              !  + hex 800
CALL ibeos (dvm,v)
LET wrt$ = "123" & chr$(10)      !  Data bytes to
                                 !  be written.
                                 !  eos character
                                 !  is the last byte
CALL ibwrt (dvm, wrt$)
```

IBEOS (continued) IBEOS

Board Function Examples:

1. Program the interface board `brd0` to terminate a read on detection of the linefeed character (hex 0A) that is expected to be received within 200 bytes.

```
DECLARE integer v, EOSV
LET EOSV = 10
'
'
'
'
LET v = EOSV + 1024             !  + hex 400
CALL ibeos (brd0, v)
!  Assume board has been addressed; do board read.
LET rd$ = space$(200)
CALL ibrd (brd0, rd$)
!  The END bit in ibsta is set if the read terminated
!  on the eos character.  The value of ibcnt shows
the
!  number of bytes received.
```

2. To program the interface board `brd0` to terminate read operations on the 8-bit value hex 82 rather than the 7-bit character hex 10, change EOSV and `v` in Example 1.

```
LET EOSV = 130                  !  hex 82
'
'
'
'
LET v = EOSV + 5120             !  + hex 1400
'
'
```

IBEOS (continued) IBEOS

3. To disable read termination on receiving the EOS character for operations involving the interface board `brd0`, change `v` in Example 1.

```
'
'
'
'
LET v = EOSV
```

4. Send END when the linefeed character is written for operations involving the interface board `brd0`.

```
DECLARE integer v, EOSV
EOSV = 10                    !  EOS info for ibeos.
'
'
'
'
LET v = EOSV + 2048          !  + hex 800
CALL ibeos (brd0, v)
!  Assume the board has been addressed; do board
write.
LET wrt$ = "123" & chr$(10)
CALL ibwrt (brd0, wrt$)
```

5. To send END with linefeeds and to terminate reads on linefeeds for operations involving the interface board `BRD0`, change `v` in Example 4.

```
'
'
LET v = EOSV + 3072           !  + hex C00
```

IBEOT IBEOT

Purpose: Enable or disable automatic END termination message on write operations.

Format: `CALL ibeot (ud,v)`

`ud` is a device or an interface board. If `v` is non-zero, the END message is sent automatically with the last byte of each write operation. If `v` is zero, END is not automatically sent. `ibeot` is needed only to alter the value from the configuration setting. (In the default configuration, this feature is enabled.)

The END message is the assertion of the GPIB EOI signal. If the automatic END termination message is enabled, it is not necessary to use the EOS character to identify the last byte of a data string. `ibeot` is used primarily to send variable length binary only data.

The sending of END with the EOS character is determined by the `ibeos` function and is not affected by the `ibeot` function.

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

When `ibeot` is called and an error does not occur, `iberr` is returned with a one if automatic END message was previously enabled, or with a zero if it was previously disabled.

Device IBEOT Function

If `ud` is a device, the END termination message method that is selected is used on all device I/O write operations to that device.

Board IBEOT Function

If `ud` is an interface board, the END termination message method that is selected is used on all board I/O write operations, regardless to which device is written.

Refer also to *IBEOS* and Table 1-2.

IBEOT (continued) IBEOT

Device Function Example:

Send the END message with the last byte of all subsequent writes to the device `plotter`.

```
DECLARE integer v, plotter
LET plotter$ = "DEV5"
CALL ibfind (plotter$, plotter)
LET v = 1                !  Enable sending of END.
CALL ibeot (plotter, v)
!  It is assumed that wrt$ contains the data to be
!  written to the GPIB.
CALL ibwrt (plotter, wrt$)
```

Board Function Examples:

1. Stop sending END with the last byte for calls directed to the interface board `brd0`.

```
DECLARE integer v
LET v = 0                !  Disable sending of END.
CALL ibeot (brd0, v)
```

2. Send the END message with the last byte of all subsequent write operations directed to the interface board `brd0`.

```
LET v = 1                !  Enable sending of END.
CALL ibeot (brd0, v)
!  It is assumed that wrt$ contains the data to be
!  written and all Listeners have been addressed.
CALL ibwrt (brd0, wrt$)
```

IBFIND IBFIND

Purpose: Open device and return the unit descriptor associated with the given name.

Format: `CALL ibfind (udname$,ud)`

`udname` is a string containing a default or configured device or board name. `ud` is a variable containing the unit descriptor returned by `ibfind`.

`ibfind` returns a number that is used in each function to identify the particular device or board that is used for that function. Calling `ibfind` is required to associate a variable name in the application program with a particular device or board name. The name used in the `udname` argument must match the default or configured device or board name. The number referred to throughout this manual as a unit descriptor is returned here in the variable `ud`.

Note: For board calls, the unit descriptor may be substituted with an integer board of zero (0) or one (1). This feature allows any of the NI-488 board functions to be used compatibly with the NI-488.2 routines described in Chapter 2, *NI-488.2 Software Routine Descriptions*.

`ibfind` performs the equivalent of `ibonl` to open the specified device or board and to initialize software parameters to their default configuration settings. Use a variable name close to the actual name of the device or board in order to simplify programming effort.

The unit descriptor is valid until `ibonl` is used to place that device or interface board offline.

If the `ibfind` call fails, a negative number is returned in place of the unit descriptor. The most probable reason for a failure is that the string argument passed into `ibfind` does not exactly match the default or configured device or board name.

IBFIND (continued) IBFIND

Device Function Example:

Assign the unit descriptor associated with the device `DEV4` (Device number 4) to `dvm`.

```
DECLARE integer dvm
LET udname$ = "DEV4"          !  Device name assigned
                              !  at configuration time.
CALL ibfind (udname$, dvm)
IF dvm < 0 GOTO 1000          !  ERROR ROUTINE
```

Board Function Example:

Assign the unit descriptor associated with the interface board `GPIB0` to the variable `brd`.

```
DECLARE integer brd0
LET udname$ = "GPIB0"         !  Board name assigned
                              !  at configuration time.
CALL ibfind (udname$, brd0)
IF brd0 < 0 GOTO 1000         !  ERROR ROUTINE
```

IBGTS IBGTS

Purpose: Go from Active Controller to Standby.

Format: `CALL ibgts (ud,v)`

`ud` is an interface board. If `v` is non-zero, the GPIB board shadow handshakes the data transfer as an Acceptor, and when the END message is detected, the GPIB board enters a Not Ready For Data (NRFD) handshake holdoff state on the GPIB. If `v` is zero, no shadow handshake or holdoff is done.

The `ibgts` function causes the GPIB board to go to the Controller Standby state and to unassert the ATN signal if it initially is the Active Controller. `ibgts` permits the GPIB Controller board to go to standby and to therefore allow transfer between GPIB devices to occur without its intervention.

If the shadow handshake option is activated, the GPIB board participates in data handshake as an Acceptor without actually Reading the data. It monitors the transfers for the END message and holds off subsequent transfers. Through this mechanism, the GPIB board can take control synchronously on a subsequent operation such as `ibcmd` or `ibrpp`.

Before performing an `ibgts` with a shadow handshake, call the `ibeos` function to establish the proper EOS character or to disable EOS detection.

The ECIC error results if the GPIB board is not CIC.

Refer also to *IBCAC*.

In the example that follows, GPIB commands and addresses are coded as printable ASCII characters.

IBGTS (continued) IBGTS

Board Function Example:

Turn the ATN line off with the `ibgts` function after unaddressing all Listeners with the Unlisten (UNL or ASCII ?) command, addressing a Talker at hex 46 (ASCII F) and addressing a Listener at hex 31 (ASCII 1) to allow the Talker to send data messages.

```
LET cmd$ = "?F1"          !  UNL MTA1 MLA2
CALL ibcmd (brd0, cmd$)
LET v = 1                 !  Listen in continuous
mode.
CALL ibgts (brd0, v)
```

IBIST IBIST

Purpose: Set or clear individual status bit for Parallel Polls.

Format: `CALL ibist (ud,v)`

`ud` is an interface board. If `v` is non-zero, the individual status bit is set. If `v` is zero, the bit is cleared.

The `ibist` function is used when the GPIB board participates in a parallel poll that is conducted by another device that is the Active Controller. The Active Controller conducts a parallel poll by asserting the EOI signal to send the Identify (IDY) message. While this message is active, each device which has been configured to participate in the poll responds by asserting a predetermined GPIB data line either true or false, depending on the value of its local ist bit. The GPIB board, for example, can be assigned to drive the DIO3 data line true if ist=1 and false if ist=0; conversely, it can be assigned to drive DIO3 true if ist=0 and false if ist=1.

The relationship between the value of ist, the line that is driven, and the sense at which the line is driven is determined by the Parallel Poll Enable (PPE) message in effect for each device. The GPIB board is capable of receiving this message either locally, via the `ibppc` function, or remotely, via a command from the Active Controller. Once the PPE message is executed, the `ibist` function changes the sense at which the line is driven during the parallel poll, and in this fashion the GPIB board can convey a 1-bit, device-dependent message to the Controller.

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

Refer also to *IBPPC*.

IBIST (continued) IBIST

Board Function Examples:

1. Set the individual status bit.

```
DECLARE integer v
LET v = 1              !  Any non-zero value will do.
CALL ibist (brd0, v)
```

2. Clear the individual status bit.

```
DECLARE integer v
LET v = 0
CALL ibist (brd0, v)
```

IBLINES IBLINES

Purpose: Return the status of the GPIB control lines.

Format: `CALL iblines (ud,clines)`

`ud` is a board descriptor. A *valid* mask is returned along with the GPIB control line state information in `clines`. The low-order byte (bits 0 through 7) of `clines` contains a mask indicating the capability of the GPIB interface board to sense the status of each GPIB control line. The upper byte (bits 8 through 15) contains the GPIB control line state information. The pattern of each byte is as follows:

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

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

Device/Board Function Example:

Test for Remote Enable (REN):

```
DECLARE integer gpib0, clines
LET udname$ = "GPIB0"
CALL ibfind (udname$, gpib0)
CALL iblines (gpib0, clines)
LET ren_i = clines
IF b_AND (ren_i, 16) <> 16 THEN GOTO 800
IF b_AND (ren_i, 4096) = 4096 THEN GOTO 900
PRINT "REN is asserted!"
STOP
800 PRINT "GPIB board is unable to monitor REN."
STOP
900 PRINT "REN is not asserted!"
STOP
END
```

IBLN IBLN

Purpose: Check for the presence of a device on the bus.

Format: `CALL ibln (ud,pad,sad,listen)`

`ud` is a board or device descriptor. `pad` (legal values are 0 to 30) is the primary GPIB address of the device. `sad` (legal values are hex 60 to hex 7e, or `NO.SAD`, or `ALL.SAD`) is the secondary GPIB address of the device.

The function `ibln` returns a non-zero value in the variable `listen` if a Listener is at the specified GPIB address.

Notice that the `sad` parameter can be a value in hex 60 to hex 7e or one of the constants `NO.SAD` or `ALL.SAD`. You can test for a Listener using only GPIB primary addressing by making `sad=NO.SAD`, or you can test all secondary addresses associated with a single primary address (a total of 31 device addresses) when you set `sad = ALL.SAD`. In this case, `ibln` sends the primary address and all secondary addresses before waiting for NDAC to settle. If the `listen` flag is true, you must search only the 31 secondary addresses associated with a single primary address to locate the Listener.

These two special constants can be used in place of a secondary address are as follows:

```
NO.SAD = 0
ALL.SAD = -1
```

If `ud` is a device, `ibln` tests for a Listener on the board associated with the given device.

Refer also to *IBDEV* and *IBFIND*.

IBLN (continued) IBLN

Device Function Example

Test for a GPIB Listener at `pad 2` and `pad hex 60`:

```
DECLARE integer ud,pad,sad,listen
LET udname$ = "DEV1"
CALL ibfind (udname$, ud)
LET pad = 2
LET sad = 96
CALL ibln (ud,pad,sad,listen)
if listen = 0
!  error: no device is at this address.
```

Board Function Example

Test for a GPIB Listener at `pad 2` and `sad hex 60`:

```
DECLARE integer ud, pad, sad, listen
LET udname$ = "GPIB0"
CALL ibfind (udname$, ud)
CALL ibln (ud, pad, sad, listen)
if listen = 0
!  error: no device is at this address.
```

IBLOC IBLOC

Purpose: Go To Local state.

Format: `CALL ibloc (ud)`

`ud` is a device or an interface board.

Unless the Remote Enable line has been unasserted with the `ibsre` function, all device functions automatically place the specified device in remote program mode. `ibloc` is used to move devices temporarily from a remote program mode to a local mode until the next device function is executed on that device.

Device IBLOC Function

`ibloc` places the device indicated in local mode by calling `ibcmd` to send the command sequence:

1. Talk address of the access board
2. Secondary address of the device, if necessary
3. Unlisten (UNL)
4. Listen address of the device
5. Secondary address of the access board
6. Go To Local (GTL)

Other command bytes can be sent as necessary.

Board IBLOC Function

If `ud` is an interface board, the board is placed in a local state by sending the local message Return To Local (RTL), if it is not locked in remote mode. The LOK bit of the status word indicates whether the board is in a lockout state. The `ibloc` function is used to simulate a front panel RTL switch when the computer is used as an instrument.

IBLOC (continued) IBLOC

Device Function Example:

Return the device `dvm` to local state.

```
DECLARE integer dvm
CALL ibloc (dvm)
```

Board Function Example:

Return the interface board `brd0` to local state.

```
DECLARE integer brd0
CALL ibloc (brd0)
```

IBONL IBONL

Purpose: Place the device or interface board online or offline.

Format: `CALL ibonl (ud,v)`

`ud` is a device or an interface board. If `v` is non-zero, the device or interface board is enabled for operation (online). If `v` is zero, it is reset (offline).

After a device or an interface board is taken offline, the handle (`ud`) is no longer valid. Before accessing the board or device again, you must re-execute an `ibfind` or `ibdev` call to open the board or device.

Calling `ibonl` with `v` non-zero restores the default configuration settings of a device or interface board.

Device Function Examples:

1. Disable the device `plotter`.

```
DECLARE integer v
LET v = 0
CALL ibonl (plotter, v)
```

2. Enable the device `plotter` after taking it offline temporarily.

```
DECLARE integer v
LET udname$ = "PLOTTER"     !  Device name assigned at
                            !  configuration time.
CALL ibfind (udname$, plotter)
!  ibonl with v non-zero is automatically
!  performed as part of ibfind.
```

IBONL (continued) IBONL

3. Reset the configuration settings of the device `plotter` to their default settings.

```
DECLARE integer v
LET v = 1
CALL ibonl (plotter, v)
```

Board Function Examples:

1. Disable the interface board `brd0`.

```
LET  v = 0
CALL ibonl (brd0, v)
```

2. Enable the interface board `brd0` after taking it offline temporarily.

```
DECLARE integer brd0
LET udname$ = "GPIB0"    !  Board name assigned at
                         !  configuration time.
CALL ibfind (udname$, brd0)
!  ibfind automatically places board online.
```

3. Reset the configuration settings of the interface board `brd0` to their default settings.

```
DECLARE integer
LET v = 1
CALL ibonl (brd0, v)
```

IBPAD IBPAD

Purpose: Change Primary Address.

Format: `CALL ibpad (ud,v)`

`ud` is a device or an interface board. `v` is the primary GPIB address. `ibpad` is needed only to alter the configuration setting.

There are 31 valid GPIB addresses, ranging from 0 to hex 1E; that is, the lower five bits of `v` are significant and they must not all be ones. An EARG error results if the value of `v` is not in this range.

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

When `ibpad` is called and an error does not occur, the previous primary address is stored in `iberr`.

Device IBPAD Function

If `ud` is a device, `ibpad` determines the talk and listen addresses based on the value of `v`. A device listen address is formed by adding hex 20 to the primary address; the talk address is formed by adding hex 40 to the primary address. A primary address of hex 10 corresponds to a listen address of hex 30 and a talk address of hex 50. The actual GPIB address of any device is set within that device, either with hardware switches or a software program. Refer to the device documentation for instructions.

Board IBPAD Function

If `ud` is a board, `ibpad` programs the board to respond to the address indicated by `v`.

Refer also to *IBSAD* and *IBONL*.

IBPAD (continued) IBPAD

Device Function Example:

Change the primary GPIB address of `plotter` to hex A

```
DECLARE integer v
LET v = 10            !  Lower 5 bits of GPIB
address.
CALL ibpad (plotter, v)
```

Board Function Example:

Change the primary GPIB address of the board `brd0` to hex 7.

```
DECLARE integer v
LET v = 7
CALL ibpad (brd0, v)
```

IBPCT IBPCT

Purpose: Pass control.

Format: `CALL ibpct (ud)`

`ud` is a device.

The `ibpct` function passes CIC authority to the specified device from the access board assigned to that device. The board automatically goes to Controller Idle State (CIDS). The function assumes that the device has Controller capability.

`ibpct` calls the board `ibcmd` function to send the following commands:

- Unlisten (UNL)
- Listen address of the access board
- Talk address of the device
- Secondary address of the device, if applicable
- Take Control (TCT)

Other command bytes can be sent as necessary.

Refer to *IBCMD* for additional information.

Device Function Example:

Pass control to the device `ibmxt`.

```
DECLARE integer ibmxt
CALL ibpct (ibmxt)
```

IBPPC IBPPC

Purpose: Parallel Poll Configure.

Format: `CALL ibppc (ud,v)`

`ud` is a device or an interface board. `v` must be a valid parallel poll enable/disable command or zero (0).

`ibppc` returns the previous value of `v` in `iberr` if an error does not occur.

Device IBPPC Function

If `ud` is a device, the `ibppc` function enables or disables the device from responding to parallel polls.

`ibppc` calls the board `ibcmd` function to send the following commands:

- Talk address of the access board
- Unlisten
- Listen address of the device
- Secondary address of the device, if applicable
- Parallel Poll Configure (PPC)
- Parallel Poll Enable (PPE) or Disable (PPD)

Other command bytes are sent if necessary.

Each of the 16 PPE messages specifies the GPIB data line (DIO1 through DIO8) and sense (one or zero) that the device must use when responding to a parallel poll. The assigned message is interpreted by the device along with the current value of the individual status (ist) bit to determine if the selected line is driven true or false. For example, if the PPE=hex 64, DIO5 is driven true if ist=0 and false if ist=1, and if PPE=hex 68, DIO1 is driven true if ist=1 and false if ist=0. Any PPD message or zero value cancels the PPE message in effect. You must know which PPE and PPD messages are sent and determine what the responses indicate.

IBPPC (continued) IBPPC

Board IBPPC Function

If `ud` is an interface board, the board itself is programmed to respond to a parallel poll by setting its Local Poll Enable (LPE) message to the value of `v`.

Refer also to *IBCMD* and *IBIST*.

Device Function Examples:

1. Configure the device `dvm` to respond to a parallel poll by sending data line DIO5 true (ist=0).

```
DECLARE integer v
LET v = 100                    !  hex 64
CALL ibppc (dvm, v)
```

2. Configure the device `dvm` to respond to a parallel poll by sending data line DIO1 true (ist=1).

```
DECLARE integer v
LET v = 104                    !  hex 68
CALL ibppc (dvm, v)
```

3. Cancel the parallel poll configuration of the device `dvm.`

```
DECLARE integer v
LET v = 112                    !  hex 70
CALL ibppc (dvm, v)
```

Board Function Example:

Configure the interface board `brd0` to respond to a parallel poll by sending data line DIO5 true (ist=0).

```
DECLARE integer v
LET v = 100                    !  hex 64
CALL ibppc (brd0, v)
```

IBRD IBRD

Purpose: Read data from a device to a string.

Format: `CALL ibrd (ud,rd$)`

`ud` is a board or a device. `rd$` is the storage buffer for data.

`ibrd` terminates when one of the following events occurs:

- The allocated buffer becomes full.
- An error is detected.
- The time limit is exceeded.
- An END message is detected.
- An EOS character is detected (if this option is enabled).

Transfer count may be less than expected if any of these terminating events, except for the first event, occurs.

When `ibrd` completes, `ibsta` holds the latest device status, `ibcnt` is the 16-bit representation of the number of bytes read, and, if the ERR bit in `ibsta` is set, `iberr` is the first error detected.

Device IBRD Function

If `ud` is a device, the device is addressed to talk and the access board is addressed to listen. Then the data is read from the device.

Board IBRD Function

If `ud` is an interface board, the `ibrd` function reads from a GPIB device that is assumed to be already properly addressed by the CIC. In addition to the termination conditions previously listed, a board `ibrd` function also terminates if a Device Clear (DCL) or Selected Device Clear (SDC) command is received from the CIC.

IBRD (continued) IBRD

If the access board is Active Controller, the board is placed in Standby Controller state with ATN off even after the operation completes. If the access board is not Active Controller, `ibrd` commences immediately.

If the board is CIC, the `ibcmd` function must be used prior to `ibrd` to address a device to talk and the board to listen.

An EADR error results if the board is CIC but has not been addressed to listen with the `ibcmd` function. An EABO error results if, for any reason, `ibrd` does not complete within the time limit.

Device Function Example:

Read 100 bytes of data from a device.

```
DECLARE integer brd0, pad, sad, tmo
DECLARE integer eot, eos, ud
LET brd0 = 0
LET pad = 10
LET sad = 0
LET tmo = 15
LET eot = 1
LET eos = 0
CALL ibdev (brd0, pad, sad, tmo, eot, eos, ud)
LET rd$ = repeat$(" ",100)
CALL ibrd (ud, rd$)
```

Board Function Examples:

1. Read 100 bytes of data from a device at talk address hex 4C (ASCII L). The listen address of the board is hex 20 (ASCII space).

```
DECLARE integer brd0
LET brd0$ = "GPIB0"
CALL ibfind (brd0$, brd0)
LET cmd$ = "? L"            !  UNL MLA TAD
CALL ibcmd (brd0, cmd$)
LET rd$ = repeat$(" ",56)
CALL ibrd (brd0, rd$)
```

IBRD **(continued)** **IBRD**

2. To terminate the read on an EOS character, see the *IBEOS* board function example.

3. To enable automatic byte swapping of binary integer data, see the *IBCONFIG* board function example.

IBRDA IBRDA

Purpose: Read data asynchronously to string.

Format: `CALL ibrda (ud,rd$)`

`ud` is a device or an interface board. `rd$` identifies the storage buffer for data bytes that are read from the GPIB.

The `ibrda` function reads up to 132 bytes of data from a GPIB device. The maximum string length can be increased to a value less than 32767.

`ibrda` is used in place of `ibrd` when the application program must perform other functions while processing the GPIB I/O operation. `ibrda` returns immediately after starting the I/O operation.

The three asynchronous I/O calls (`ibcmda, ibrda`, and `ibwrta`) are designed to allow an application to perform other functions (non-GPIB functions) while processing the I/O. Once the asynchronous I/O call has been initiated, further GPIB calls involving the device or access board are not allowed until the I/O has completed and the GPIB driver and the application have been resynchronized.

Resynchronization can be accomplished by using one of the following three functions:

Note: Resynchronization is only successful if the `ibsta` returned contains CMPL.

- `ibwait` (mask contains CMPL) - The driver and application are synchronized.
- `ibstop` - The asynchronous I/O is canceled, and the driver and application are synchronized.
- `ibonl` - The asynchronous I/O is canceled, the interface has been reset, and the driver and application are synchronized.

The only other GPIB call that is allowed during asynchronous I/O is the `ibwait` function (mask is arbitrary). Any other GPIB call involving the device or access board returns the EOIP error.

IBRDA (continued) IBRDA

Device IBRDA Function

If `ud` is a device, the device is addressed to talk and the access board is addressed to listen. Then the data is read from the device. Other command bytes may be sent as necessary.

Board IBRDA Function

If `ud` is an interface board, the `ibrda` function attempts to read from a GPIB device that is assumed to be already properly addressed.

If the board is CIC, the `ibcmd` function must be called prior to `ibrda` to address the device to talk and the board to listen. Otherwise, the actual CIC must perform the addressing.

If the board is Active Controller, the board is first placed in Standby Controller state with ATN off even after the read operation completes. If the board is not the Active Controller, the read operation commences immediately.

An EADR error results if the interface board is CIC but has not addressed itself to listen with the `ibcmd` function.

Device Function Example:

Read 56 bytes of data from the device `tape` while performing other processing.

```
!  Perform device read.
LET rd$ = repeat$(" ",56)
CALL ibrda (tape, rd$)
LET mask = 16640              !  TIMO CMPL
!  Perform other processing here then
!  wait for I/O completion or a timeout.
CALL ibwait (tape, mask)
!  Check ibsta to see what the read terminated:
!  on CMPL, END, TIMO, or ERR.  (not done here)
```

IBRDA (continued) IBRDA

Board Function Examples:

1. Read 56 bytes of data from a device at talk address hex 4C (ASCII L) and then unaddress it (the GPIB board listen address is hex 20 or ASCII space).

```
!  Perform addressing in preparation for board read.
LET cmd$ = "? L"              !  UNL MLA TAD
CALL ibcmd (brd0, cmd$)
!  Perform board read.
LET rd$ = space$(56)
CALL ibrda (brd0, rd$)
!  Perform other processing here, then wait for
!  I/O completion or a timeout.
LET mask = hex 16640          !  TIMO CMPL
CALL ibwait (brd0, mask)
!  ibsta shows how the read terminated: on
!  CMPL, END, TIMO, or ERR.
```

2. To terminate the read on an EOS character, see the *IBEOS Board Function Examples.*

3. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example.*

IBRDF IBRDF

Purpose: Read data from GPIB into file.

Format: `CALL ibrdf (ud,flname$)`

`ud` is a device or an interface board. `flname` is the filename under which the data is stored. `flname` can be up to 50 characters long, including a drive and path designation.

`ibrdf` automatically opens the file as a binary file (not as a character file). If the file does not exist, `ibrdf` creates it. On exit, `ibrdf` closes the file.

An EFSO error results if it is not possible to open, create, seek, write, or close the specified file.

The `ibrdf` function terminates on any of the following events:

- An error is detected.
- The time limit is exceeded.
- An END message is detected.
- An EOS character is detected (if this option is enabled).
- A Device Clear (DCL) or Selected Device Clear (SDC) command is received from another device which is the CIC.

After termination, `ibcnt` is the number of bytes read.

When the device `ibrdf` function returns, `ibsta` holds the latest device status, `ibcnt` is the 16-bit representation of the number of bytes read, and if the ERR bit in `ibsta` is set, `iberr` is the first error detected.

IBRDF (continued) IBRDF

Device IBRDF Function

If `ud` is a device, the same board functions as the device `ibrd` function are performed automatically. The `ibrdf` function terminates on similar conditions as `ibrd`.

Board IBRDF Function

If `ud` is an interface board, the board `ibrd` function reads from a GPIB device that is assumed to be already properly addressed.

An EADR error results if the board is CIC but has not been addressed to listen with the `ibcmd` function. An EABO error results if, for any reason, the read operation does not complete within the time limit. An EABO error also results if the device that is to talk is not addressed and/or the operation does not complete within the time limit for whatever reason.

Device Function Example:

Read data from the device `rdr` into the file `RDGS` on disk drive `B`.

```
LET flname$ = "B:RDGS"
CALL ibrdf (rdr, flname$)
!  ibsta and ibcnt show the results of the
!  read operation.
```

IBRDF (continued) IBRDF

Board Function Example:

Read data from a device at talk address hex 4C (ASCII L) to the file RDGS on the current disk drive and then unaddress it (the GPIB board listen address is hex 20 or ASCII space).

```
!  Perform addressing in preparation for board read.
LET cmd$ = "?L "              !  UNL TAD MLA
CALL ibcmd (brd0, cmd$)
!  Perform board read.
LET flname$ = "RDGS"
CALL ibrdf (brd0, flname$)
!  ibsta and ibcnt show the results of the read
!  operation (not done here).
```

IBRDI IBRDI

Purpose: Read data to integer array.

Format: `CALL ibrdi (ud,iarr(1),cnt)`

`ud` is a device or an interface board. `iarr` is an integer array into which data is read from the GPIB. `cnt` is the maximum number of bytes to be read.

`ibrdi` is similar to the `ibrd` function, which reads data into a character string variable. As the data is read, each byte pair is treated as an integer and stored in `iarr`.

Unlike `ibrd`, `ibrdi` stores the data directly into an integer array. No integer conversion of the data is needed for arithmetic operations.

Refer to *IBRD* and tothe section titled *NKR BASIC NI-488 I/O Calls and Functions* in Chapter 1, *General Information*.

Device Function Example:

Read bytes of data from the device `tape` and store in the integer array `rd`.

```
DECLARE integer cnt, rd(256), tape, pad, sad
DECLARE integer tmo, eot, eos, brd
LET brd = 0
LET pad = 6
LET sad = 0
LET tmo = 14
LET eot = 1
LET eos = 0
CALL ibdev (brd, pad, sad, tmo, eot, eos, tape)
!  cnt is equal to array size multiplied by 2.
LET cnt=512
CALL ibrdi (tape, rd(1), cnt)
```

IBRDI (continued) IBRDI

Board Function Examples:

1. Read 56 bytes of data into the integer array `rd` from a device at talk address hex 4C (ASCII L). (The GPIB board listen address is hex 20 or ASCII space.)

```
DECLARE integer cnt, rd(28)
!  Perform addressing in preparation for board read.
LET cmd$ = "? L"              !  UNL MLA TAD
CALL ibcmd (brd0, cmd$)
!  cnt is equal to array size multiplied by 2.
LET cnt = 56
CALL ibrdi (brd0, rd(1), cnt)
!  ibsta shows how the read terminated:
!  CMPL, END, TIMO, or ERR.  Data is stored in rd().
```

2. To terminate the read on an EOS character, see *IBEOS* examples.

3. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

IBRDIA IBRDIA

Purpose: Read data asynchronously to integer array.

Format: `CALL ibrdia (ud,iarr(1),cnt)`

`ud` is a device or an interface board. `iarr` is an integer array into which data is read asynchronously from the GPIB. `cnt` specifies the maximum number of bytes to be read.

`ibrdia` is similar to the `ibrda` function, which reads data into a character string variable. As the data is read, each byte pair is treated as an integer and stored in `iarr`.

Unlike `ibrda`, `ibrdia` stores the data directly into an integer array. No integer conversion of the data is needed for arithmetic operations.

Refer to *IBRDA* and tothe section titled *NKR BASIC NI-488 I/O Calls and Functions* in Chapter 1, *General Information*.

Device Function Example:

Read 56 bytes of data into the integer array `rd` from the device `tape` while performing other processing.

```
DECLARE integer cnt, rd(28)
!  cnt is equal to array size multiplied by 2.
LET cnt = 56
CALL ibrdia (tape, rd(0), cnt)
LET mask = 16640              !  TIMO CMPL
!  Perform other processing here then wait for I/O
!  completion or a timeout.
CALL ibwait (tape, mask)
!  Check ibsta to see what the read terminated:
!  on CMPL, END, TIMO, or ERR (not done here).
```

IBRDIA (continued) IBRDIA

Board Function Examples:

1. Read 56 bytes of data into the integer array `rd` from a device at talk address hex 4C (ASCII L). (The GPIB board listen address is hex 20 or ASCII space.)

```
DECLARE integer cnt, rd(28), mask
!  Perform addressing in preparation for board read.
LET cmd$ = "? L"              !  UNL MLA TAD
CALL ibcmd (brd0, cmd$)
!  Perform board read.
!  cnt is equal to array size multiplied by 2.
LET cnt = 16640
CALL ibrdia (brd0, rd(1), cnt)
LET mask = 16640              !  TIMO CMPL
!  Perform other processing here, then wait for I/O
!  completion or a timeout.
CALL ibwait (brd0, mask)
!  ibsta shows how the read terminated: CMPL, END,
!  TIMO, or ERR.  If CMPL or ERR are not set,
!  continue processing.
LET ibsta_i = ibsta
IF b_ AND (ibsta_i, -32512) <> 0 GOTO 200
!  Properly terminate the asynchronous I/O
LET mask = 16640
CALL ibwait (brd0, mask)
!  Data is stored in rd$.
```

2. To terminate the read on an EOS character, see the *IBEOS Board Function Example*.

3. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example.*

IBRPP IBRPP

Purpose: Conduct a Parallel Poll.

Format: `CALL ibrpp (ud,ppr)`

`ud` is a device or an interface board. `ppr` stores the parallel poll response.

Device IBRPP Function

If `ud` is a device, all devices on its GPIB are polled in parallel using the access board of that device. This is done by executing the board `ibrpp` function with the appropriate access board specified.

Board IBRPP Function

If `ud` is a board, the `ibrpp` function causes the identified board to conduct a parallel poll of previously configured devices by sending the IDY message (ATN and EOI both asserted) and reading the response from the GPIB data lines.

An ECIC error results if the GPIB board is not CIC. If the GPIB board is Standby Controller, it takes control and asserts ATN (becomes Active) prior to polling. It remains Active Controller afterward.

In the examples that follow, some of the GPIB commands and addresses are coded as printable ASCII characters. The simplest means of specifying values is to use printable ASCII characters to represent values. When possible, ASCII characters should be used. Refer to Appendix A for conversions of numeric values to ASCII characters.

Some commands relevant to parallel polls are shown in Table 3-4.

IBRPP (continued) IBRPP

Table 3-4. Parallel Poll Commands

Command	Hex Value	Meaning
PPC	05	Parallel Poll Configure
PPU	15	Parallel Poll Unconfigure
PPE	60	Parallel Poll Enable
PPD	70	Parallel Poll Disable

Parallel poll constants are defined in the file `decl.b`.

Device Function Example:

Remotely configure the device `lcrmtr` to respond positively on DI03 if its individual status bit is 1, and then parallel poll all configured devices.

```
DECLARE integer v, ppr
LET v = 106                    !  hex 6A
CALL ibppc (lcrmtr, v)
CALL ibrpp (lcrmtr, ppr)
```

IBRPP (continued) IBRPP

Board Function Examples:

1. Remotely configure the device `brd0` at listen address hex 23 (ASCII #) to respond positively on DIO3 if its individual status bit is 1, and then parallel poll all configured devices.

```
DECLARE integer ppr
!  Send LAD, PPC, PPE, and UNL.
LET cmd$ = "#" & chr$(5) & "j?"
CALL ibcmd (brd0, cmd$)
CALL ibrpp (brd0, ppr)
```

2. Disable and unconfigure all GPIB devices from parallel polling using the PPU (hex 15) command.

```
DECLARE integer ppr
LET cmd$ = chr$(21)           !  PPU
CALL ibcmd (brd0, cmd$)
```

IBRSC IBRSC

Purpose: Request or release System Control.

Format: `CALL ibrsc (ud,v)`

`ud` is an interface board. If `v` is non-zero, functions requiring System Controller capability are subsequently allowed. If `v` is zero (0), functions requiring System Controller capability are not allowed.

The `ibrsc` function is used to enable or disable the capability of the GPIB board to send the Interface Clear (IFC) and Remote Enable (REN) messages to GPIB devices using the `ibsic` and `ibsre` functions, respectively. The interface board must not be System Controller to respond to IFC sent by another Controller.

In most applications, the GPIB board is always the System Controller. In other applications, the GPIB board is never the System Controller. In either case, the `ibrsc` function is used only if the computer is not going to be System Controller for the duration of the program execution. While the IEEE 488 standard does not specifically allow schemes in which System Control can be passed dynamically from one device to another, the `ibrsc` function would be used in such a scheme.

When `ibrsc` is called and an error does not occur, `iberr` is set to one (1) if the interface board was previously System Controller and zero (0) if it was not.

Refer also to Table 1-2.

Board Function Example:

Request to be System Controller if the interface board `brd0` is not currently so designated.

```
DECLARE integer v
LET v = 1               !  Any non-zero value will do.
CALL ibrsc (brd0, v)
```

IBRSP IBRSP

Purpose: Return serial poll byte.

Format: `CALL ibrsp (ud,spr)`

`ud` is a device. `spr` stores the serial poll response.

The `ibrsp` function is used to serial poll one device and obtain its status byte or to obtain a previously stored status byte. If bit 6 (the hex 40 bit) of the response is set, the device is requesting service.

When the automatic serial polling feature is enabled, the specified device may have been polled previously. If it has been polled and a positive response was obtained, the RQS bit of `ibsta` is set on that device. In this case, `ibrsp` returns the previously acquired status byte. If the RQS bit of `ibsta` is not set during an automatic poll, it serial polls the device.

When a poll is actually conducted, the specific sequence of events is as follows:

1. Unlisten (UNL)
2. Controllers Listen Address
3. Secondary address of the access board, if applicable
4. Serial Poll Enable (SPE)
5. Talk address of the device
6. Secondary address of the device, if applicable
7. Read serial poll response byte from device
8. Serial Poll Disable (SPD)
9. Other command bytes may be sent as necessary

IBRSP (continued) IBRSP

The response byte `spr`, except the RQS bit, is device specific. For example, the polled device might set a particular bit in the response byte to indicate that it has data to transfer and another bit to indicate a need for reprogramming. Consult the device documentation for interpretation of the response byte.

Refer to *IBCMD* and *IBRD* for additional information.

Device Function Example:

Obtain the Serial Poll Response (`spr`) byte from the device `tape`.

```
DECLARE integer spr
CALL ibrsp (tape, spr)
!  The application program would then analyze the
!  response in spr.
```

IBRSV IBRSV

Purpose: Request service and/or set or change the serial poll status byte.

Format: `CALL ibrsv (ud,v)`

`ud` is an interface board. `v` is the status byte that the GPIB board provides when serial polled by another device that is the GPIB CIC. If bit 6 (the hex 40 bit) is set, the GPIB board additionally requests service from the Controller by asserting the GPIB SRQ line.

The `ibrsv` function is used to request service from the Controller using the Service Request (SRQ) signal and to provide a system-dependent status byte when the Controller serial polls the GPIB board.

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

Refer also to Table 1-2.

Board Function Examples:

1. Set the Serial Poll status byte to hex 41, which simultaneously requests service from an external CIC.

```
DECLARE integer v, stb
LET stb = 1
LET v = 65 OR stb              !  Assert SRQ
CALL ibrsv (brd0, v)
```

2. Change the status byte without requesting service.

```
DECLARE integer stb
LET stb = 35                   !  New status byte
                               !  value, hex 23.
CALL ibrsv (brd0, stb)
```

IBSAD IBSAD

Purpose: Change or disable Secondary Address.

Format: `CALL ibsad (ud,v)`

`ud` is a device or an interface board. If `v` is a number between hex 60 and hex 7E, that number becomes the secondary GPIB address of the device or interface board. If `v` is hex 7F or zero (0), secondary addressing is disabled. `ibsad` is needed only to alter the secondary address value from its configuration setting.

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

When `ibsad` is called and an error does not occur, the previous secondary address is stored in `iberr`.

Device IBSAD Function

If `ud` is a device, the function enables or disables extended GPIB addressing for the device. When secondary addressing is enabled, the specified secondary GPIB address of that device is sent automatically in subsequent device I/O functions.

Board IBSAD Function

If `ud` is an interface board, the `ibsad` function enables or disables extended GPIB addressing and, when enabled, assigns the secondary address of the GPIB board.

Refer also to *IBPAD* and *IBONL*.

Device Function Examples:

1. Change the secondary GPIB address of the device `plotter` from its current value to hex 6A.

```
DECLARE integer v
LET v = 106
CALL ibsad (plotter, v)
```

IBSAD (continued) IBSAD

2. Disable secondary addressing for the device `dvm`.

```
DECLARE integer v
LET v = 0                !  0 or hex 7F can be used.
CALL ibsad (dvm, v)
```

Board Function Examples:

1. Change the secondary GPIB address of the interface board `brd0` from its current value to hex 6A.

```
DECLARE integer v
LET v = hex 6A
CALL ibsad (brd0, v)
```

2. Disable secondary addressing for the interface board `brd0`.

```
LET v = 0                !  0 or hex 7F can be used.
CALL ibsad (brd0, v)
```

IBSIC IBSIC

Purpose: Send interface clear for 100 µsec.

Format: `CALL ibsic (ud)`

`ud` is an interface board. `ibsic` must be used at the beginning of a program if board functions are used.

The `ibsic` function asserts the IFC signal for at least 100 µsec, provided the GPIB board has System Controller capability. This action initializes the GPIB, makes the interface board CIC and Active Controller with ATN asserted, and is generally used when a bus fault condition is suspected.

The IFC signal resets only the GPIB interface functions of bus devices and not the internal device functions. Device functions are reset with the Device Clear (DCL) and Selected Device Clear (SDC) commands. To determine the effect of these messages, consult the device documentation.

The ESAC error occurs if the GPIB board does not have System Controller capability.

Refer also to *IBRSC*.

Board Function Example:

At the beginning of a program, initialize the GPIB and become CIC and Active Controller.

```
CALL ibsic (brd0)
```

IBSRE IBSRE

Purpose: Set or clear the Remote Enable line.

Format: `CALL ibsre (ud,v)`

`ud` is an interface board. If `v` is non-zero, the Remote Enable (REN) signal is asserted. If `v` is zero (0), the signal is unasserted.

The `ibsre` function turns the REN signal on and off. REN is used by devices to select between local and remote modes of operation. REN enables the remote mode. A device does not actually enter remote mode until it receives its listen address.

The ESAC error occurs if the GPIB board is not System Controller.

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

Refer also to *IBRSC* and Table 1-2.

Board Function Examples:

1. Place the device at listen address hex 23 (ASCII #) in remote mode.

```
DECLARE integer v
LET v = 1               !  Any non-zero value will do.
CALL ibsre (brd0, v)
LET cmd$ = "#"          !  LAD
CALL ibcmd (brd0, cmd$)
```

IBSRE (continued) IBSRE

2. To exclude the local ability of the device to return to local mode, send the Local Lockout (LLO or hex 11) command or include it in the command string in Example 1.

```
LET cmd$ = chr$(17)
CALL ibcmd (brd0, cmd$)
```

or

```
LET cmd$ = "#" & chr$(17)
CALL ibcmd (brd0, cmd$)
```

3. Return all devices to local mode.

```
DECLARE integer v
LET v = 0               !  Set REN to false
CALL ibsre (brd0, v)
```

IBSTOP IBSTOP

Purpose: Abort asynchronous operation.

Format: `CALL ibstop (ud)`

`ud` is a device or an interface board.

`ibstop` terminates any asynchronous read, write, or command operation and then resynchronizes the application with the driver.

If there is an aysnchronous I/O operation in progress, the ERR bit in the status word is set and an EABO error is returned.

Device IBSTOP Function

If `ud` is a device, `ibstop` attempts to terminate any unfinished asynchronous I/O device function to that device.

Board IBSTOP Function

If `ud` is a board, `ibstop` attempts to terminate any unfinished asynchronous I/O operation that had been started with that board.

Device Function Example:

Stop any asynchronous operations associated with the device `rdr`.

```
CALL ibstop (rdr)
```

Board Function Example:

Stop any asynchronous operations associated with the interface board `brd0`.

```
CALL ibstop (brd0)
```

IBTMO IBTMO

Purpose: Change or disable time limit.

Format: `CALL ibtmo (ud,v)`

ud is a device or an interface board. v is a code specifying the time limit as follows:

Table 3-5. Timeout Code Values

Value of v	Minimum Timeout
0	disabled
1	10 µsec
2	30 µsec
3	100 µsec
4	300 µsec
5	1 msec
6	3 msec
7	10 msec
8	30 msec
9	100 msec
10	300 msec
11	1 sec
12	3 sec
13	10 sec
14	30 sec
15	100 sec

(continues)

IBTMO (continued) IBTMO

Table 3-5. Timeout Code Values (continued)

Value of v	Minimum Timeout
16	300 sec
17	1000 sec

Note: If `v` is zero, no limit is in effect.

`ibtmo` is needed only to alter the value from its configuration setting.

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

The `ibtmo` function changes the length of time that the following functions wait for the embedded I/O operation to finish or for the specified event to occur before returning with a timeout indication:

- `ibcmd`
- `ibrd`
- `ibrdi`
- `ibwrt`
- `ibwrti`

The `ibtmo` function also changes the length of time that device functions wait for commands to be accepted. If a device does not accept commands within the time limit, the EBUS error will be returned.

When `ibtmo` is called and an error does not occur, the previous timeout code value is stored in `iberr`.

IBTMO (continued) IBTMO

Device IBTMO Function

If `ud` is a device, the new time limit is used in subsequent device functions directed to that device.

Board IBTMO Function

If `ud` is a board, the new time limit is used in subsequent board functions directed to that board.

Refer also to *IBWAIT* and Table 1-2.

Device Function Example:

Change the time limit for calls involving the device `tape` to approximately 300 msec.

```
DECLARE integer v, tape
LET tape$ = "DEV9"
CALL ibfind (tape$, tape)
LET v = 10
CALL ibtmo (tape, v)
```

Board Function Example:

Change the time limit for calls directed to the interface board `brd0` to approximately 10 msec.

```
DECLARE integer v
LET v = 7
CALL ibtmo (brd0, v)
```

IBTRAP IBTRAP

Purpose: Alter Applications Monitor trap and display modes.

Format: `CALL ibtrap (mask,mode)`

`mask` is a bit mask with the same bit assignments as `ibsta`. Each `mask` bit is set to be trapped and/or recorded (depending on the value of `mode`) when the corresponding bit appears in the status word after a GPIB call. If all the bits are set, then every GPIB call except `ibfind` is trapped.

`mode` determines whether the recording and trapping occur. The valid mode values are listed in Table 3-6.

Table 3-6. IBTRAP Mode

Value	Effect
1	Turn monitor off. No recording or trapping occurs.
2	Turn record on. All calls are recorded but no trapping occurs.
3	Turn record and trap on. All calls are recorded and the monitor is displayed whenever a trap condition occurs.

If an error occurs during this call, the ERR bit of `ibsta` is set and `iberr` is one of the values listed in Table 3-7. Otherwise, `iberr` contains the previous `mask` value.

Table 3-7. IBTRAP Errors

Value	Explanation
1	Applications Monitor not installed.
2	Invalid monitor mode.
3	`ibtrap` not supported by installed driver.

Refer to Appendix B, *Applications Monitor*, for more information.

IBTRAP (continued) IBTRAP

Device Function Example:

Configure applications monitor to record and trap on SRQI or CMPL.

```
DECLARE integer mask, mode
LET mask = 4352           !  SRQI or CMPL (hex 1100)
LET mode = 3              !  Record and trap on
CALL ibtrap (mask, mode)
```

IBTRG IBTRG

Purpose: Trigger selected device.

Format: `CALL ibtrg (ud)`

`ud` is a device.

`ibtrg` addresses and triggers the specified device.

`ibtrg` sends the following commands:

- Talk address of access board
- Secondary address of access board, if applicable
- Unlisten
- Listen address of the device
- Secondary address of the device, if applicable
- Group Execute Trigger (GET)

Other command bytes may be sent as necessary.

Refer to *IBCMD* for additional information.

Device Function Example:

Trigger the device `analyz`.

```
CALL ibtrg (analyz)
```

IBWAIT IBWAIT

Purpose: Wait for selected event.

Format: `CALL ibwait (ud,mask)`

`ud` is a device or an interface board. `mask` is a bit mask with the same bit assignments as the status word, `ibsta`. `ibwait` is used to monitor the events selected by the bits in mask and to delay processing until any of them occurs. These events and bit assignments are shown in Table 3-8.

Table 3-8. Wait Mask Layout

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

IBWAIT (continued) IBWAIT

`ibwait` also updates `ibsta`. If `mask` = 0 or `mask` = hex 8000 (the ERR bit), the function returns immediately.

If the TIMO bit is zero (0) or the time limit is set to zero (0) with the `ibtmo` function, timeouts are disabled. Disabling timeouts should be done only when setting `mask` = 0 or when it is certain the selected event will occur; otherwise, the processor may wait indefinitely for the event to occur.

Device IBWAIT Function

If `ud` is a device, only the ERR, TIMO, END, RQS, and CMPL bits of the wait mask and status word are applicable. If automatic polling is enabled, on an `ibwait` for RQS, each time the GPIB SRQ line is asserted, the access board of the specified device serial polls all devices on its GPIB and saves the responses, until the status byte returned by the device being waited for indicates that it was the device requesting service (bit hex 40 is set in the status byte). If the TIMO bit is set, `ibwait` returns if the event does not occur within the timeout period of the device.

Board IBWAIT Function

If `ud` is a board, all bits of the wait mask and status word are applicable except RQS.

Device Function Example:

Wait indefinitely for the device `logger` to request service.

```
DECLARE integer mask
LET mask = 2048            !  RQS (hex 800)
CALL ibwait (logger, mask)
```

IBWAIT (continued) IBWAIT

Board Function Examples:

1. Wait for a service request or a timeout.

```
DECLARE integer mask
LET mask = 20480          !  TIMO SRQI (hex 5000)
CALL ibwait (ud, mask)
!  Check ibsta here to see which occurred.
```

2. Update the current status for `ibsta`.

```
DECLARE integer mask
LET mask = 0
CALL ibwait (ud, mask)
```

3. Wait indefinitely until control is passed from another CIC.

```
DECLARE integer mask
LET mask = 32             !  CIC (hex 20)
CALL ibwait (ud, mask)
```

4. Wait indefinitely until addressed to talk or listen by another CIC.

```
DECLARE integer mask
LET mask = 12             !  TACS LACS (hex 0C)
CALL ibwait (ud, mask)
```

IBWRT IBWRT

Purpose: Write data from string.

Format: `CALL ibwrt (ud,wrt$)`

`ud` is a device or an interface board. `wrt` contains the data to be sent over the GPIB.

The `ibwrt` function terminates on any of the following events:

- All bytes are transferred.
- An error is detected.
- The time limit is exceeded.
- A Device Clear (DCL) or Selected Device Clear (SDC) command is received from another device which is the CIC.

`ibcnt` is the 16-bit representation of the number of bytes read. A short count can occur on any of the above terminating events but the first.

When the device `ibwrt` function returns, `ibsta` holds the latest device status, `ibcnt` is the 16-bit representation of the number of data bytes written, and, if the ERR bit in `ibsta` is set, `iberr` is the first error detected.

Device IBWRT Function

If `ud` is a device, the device is addressed to listen and the access board is addressed to talk.

Then the data is written to the device.

Board IBWRT Function

If `ud` is an interface board, the `ibwrt` function attempts to write to a GPIB device that is assumed to be already addressed by the CIC.

If the access board is CIC, `ibcmd` must be called prior to `ibwrt` to address the device to listen and the board to talk.

IBWRT (continued) IBWRT

If the access board is Active Controller, the board is first placed in Standby Controller state with ATN off even after the write operation completes. If the access board is not the Active Controller, `ibwrt` commences immediately.

An EADR error results if the board is CIC but has not been addressed to talk with `ibcmd`. An EABO error results if, for any reason, `ibwrt` does not complete within the time limit. An ENOL error occurs if there are no Listeners on the bus when the data bytes are sent.

Note: If you want to send an EOS character at the end of your data string, you must place it there explicitly. See *Device Example 2*.

Device Function Examples:

1. Write ten instruction bytes to the device `dvm`.

```
LET wrt$ = "F3R1X5P2G0"
CALL ibwrt (dvm, wrt$)
```

2. Write five instruction bytes terminated by a carriage return and a linefeed to the device `ptr`. Linefeed is the EOS character of the device.

```
LET wrt$ = "IP2X5" & chr$(13) & chr$(10)
CALL ibwrt (ptr, wrt$)
```

Board Function Example:

Write ten instruction bytes to a device at listen address hex 2F (ASCII /) and then unaddress it (the GPIB board talk address is hex 40 or ASCII @).

```
!  Perform addressing.
LET cmd$ = "?@/"              !  UNL MTA LAD
CALL ibcmd (brd0, cmd$)
!  Perform board write.
LET wrt$ = "F3R1X5P2G0"
CALL ibwrt (brd0, wrt$)
```

IBWRTA IBWRTA

Purpose: Write data asynchronously from string.

Format: `CALL ibwrta (ud,wrt$)`

`ud` is a device or an interface board. `wrt` contains the data to be sent over the GPIB.

`ibwrta` is used in place of `ibwrt` when the application program must perform other functions while processing the GPIB I/O operation. `ibwrta` returns immediately after starting the I/O operation.

The three asynchronous I/O calls (`ibcmda`, `ibrda`, and `ibwrta`) are designed to allow an application to perform other functions (non-GPIB functions) while processing the I/O. Once the asynchronous I/O call has been initiated, further GPIB calls involving the device or access board are not allowed until the I/O has completed and the GPIB driver and the application have been resynchronized.

Resynchronization can be accomplished by using one of the following three functions:

Note: Resynchronization is only successful if the `ibsta` returned contains CMPL.

- `ibwait` (mask contains CMPL) - The driver and application are synchronized.
- `ibstop` - The asynchronous I/O is canceled, and the driver and application are synchronized.
- `ibonl` - The asynchronous I/O is canceled, the interface has been reset, and the driver and application are synchronized.

The only other GPIB call that is allowed during asynchronous I/O is the `ibwait` function (mask is arbitrary). Any other GPIB call involving the device or access board returns the EOIP error.

IBWRTA (continued) IBWRTA

Device IBWRTA Function

If `ud` is a device, the device is addressed to listen and the access board is addressed to talk. Then the data is written to the device.

Board IBWRTA Function

If `ud` is an interface board, the `ibwrta` function attempts to write to a GPIB device that is assumed to be already properly initialized and addressed by the actual CIC.

If the board is CIC, the `ibcmd` function must be called prior to `ibwrta` to address the device to listen and the board to talk.

If the board is Active Controller, the board is first placed in Standby Controller state with ATN off (even after the write operation completes). Otherwise, the write operation commences immediately.

An EADR error results if the board is CIC but has not been addressed to talk with the `ibcmd` function. The ENOL error does *not* occur if there are no Listeners.

Note: If you want to send an EOS character at the end of your data string, you must place it there explicitly.

When the device `ibwrt` function returns, `ibsta` holds the latest device status, and, if the ERR bit in ibsta is set, `iberr` is the first error detected.

IBWRTA (continued) IBWRTA

Device Function Example:

Write ten instruction bytes to the device `dvm` while performing other processing.

```
DECLARE integer mask
LET wrt$ = "F3R1X5P2G0"
CALL ibwrta (dvm, wrt$)
LET mask = 16640              !  TIMO CMPL
!  Perform other processing here then wait for I/O
!  completion or a timeout.
CALL ibwait (dvm, mask)
!  Check ibsta to see what the write terminated: on
!  CMPL, END, TIMO, or ERR
```

Board Function Example:

Write ten instruction bytes to a device at listen address hex 2F (ASCII /), while testing for a high priority event to occur, and then unaddress it (the GPIB board talk address is hex 40 or ASCII @).

```
!  Perform addressing in preparation for board write.
LET cmd$ = "?@/"              !  UNL MTA LAD
CALL ibcmd (brd0, cmd$)
!  Perform board asynchronous write.
LET wrt$ = "F3R1X5P2G0"
CALL ibwrta (brd0, wrt$)
!  Perform other processing here then wait for I/O
!  completion or a timeout.
LET mask = 16640              !  TIMO CMPL
CALL ibwait (brd0, mask)
```

IBWRTF IBWRTF

Purpose: Write data from file.

Format: `CALL ibwrtf (ud,flname$)`

`ud` is a device or an interface board. `flname` is the file name from which the data is written to the GPIB. `flname` can be up to 50 characters long, including a drive and path designation.

`ibwrtf` automatically opens the file. On exit, `ibwrtf` closes the file.

An EFSO error results if it is not possible to open, seek, read, or close the specified file.

The `ibwrtf` function operation terminates on any of the following events:

- All bytes are sent.
- An error is detected.
- The time limit is exceeded.
- A Device Clear (DCL) or Selected Device Clear (SDC) command is received from another device that is the CIC.

`ibcnt` is the 16-bit representation of the number of bytes written.

When the `ibwrtf` function returns, `ibsta` holds the latest device status, `ibcnt` is the 16-bit representation of the number of bytes written, and, if the ERR bit in `ibsta` is set, `iberr` is the first error detected.

Board IBWRTF Function

If `ud` is an interface board, the board `ibwrt` function writes to a GPIB device that is assumed to be already properly addressed.

An EADR error results if the board is CIC but has not been addressed to talk with the `ibcmd` function. An EABO error results if, for any reason, the read operation does not complete within the time limit. An ENOL error occurs if there are no Listeners on the bus when the data bytes are sent.

IBWRTF (continued) IBWRTF

Device Function Example:

Write data to the device `rdr` from the file `Y.DAT` on the current disk drive.

```
LET flname$ = "Y.DAT"
CALL ibwrtf (rdr, flname$)
```

Board Function Example:

1. Write data to the device at listen address hex 2C (ASCII ,) from the file `Y.DAT` on the current drive, and then unaddress the interface board `brd0`.

```
!  Perform addressing in preparation for board write.
LET cmd$ = "?@,"              !  UNL MTA LAD
CALL ibcmd (brd0, cmd$)
!  Perform board write.
LET flname$ = "Y.DAT"
CALL ibwrtf (brd0, flname$)
!  Unaddress the Talker and Listener.
LET cmd$ = "_?"               !  UNT UNL
CALL ibcmd (brd0, cmd$)
```

2. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

IBWRTI IBWRTI

Purpose: Write data from integer array.

Format: `CALL ibwrti (ud,iarr(1),cnt)`

`ud` is a device or an interface board. `iarr` is an integer array from which data is written to the GPIB. `cnt` specifies the maximum number of bytes to be written.

Write `cnt` bytes of data from `iarr` to the GPIB. The data, stored as 2-byte integers in `iarr`, is sent in low-byte, high-byte order to the GPIB.

`ibwrti` is similar to the `ibwrt` function, which writes data from a character string variable.

Refer to *IBWRT* and tothe section titled *NKR BASIC NI-488 I/O Calls* in Chapter 1, *General Information*. Refer also to *IBWRTIA*.

Device Function Examples:

1. Write ten instruction bytes from the integer array `wrt` to the device `dvm`.

```
DECLARE integer wrt(5), cnt
LET wrt(1) = ORD("F") + ORD("3") * 256
LET wrt(2) = ORD("R") + ORD("1") * 256
LET wrt(3) = ORD("X") + ORD("5") * 256
LET wrt(4) = ORD("P") + ORD("2") * 256
LET wrt(5) = ORD("G") + ORD("0") * 256
LET cnt = 10
CALL ibwrti (dvm, wrt(1), cnt)
```

IBWRTI (continued) IBWRTI

2. Write five instruction bytes from integer array `wrt` terminated by a carriage return and a linefeed to device `ptr`. Linefeed is the EOS character of the device.

```
DECLARE integer wrt(4), cnt
LET wrt(1) = ORD("I") + ORD("P") * 256
LET wrt(2) = ORD("2") + ORD("X") * 256
LET wrt(3) = ORD("5") + 13 * 256
LET wrt(4) = 10
LET cnt = 7
CALL ibwrti (ptr, wrt(1), cnt)
```

Board Function Example:

1. Write ten instruction bytes from the integer array `wrt` to a device at listen address hex 2F (ASCII /) and then unaddress it. (The GPIB board talk address is hex 40 or ASCII @.)

```
DECLARE integer wrt(5), cnt
!  Perform addressing.
cmd$ = "?@/"                !  UNL MTA LAD
CALL ibcmd (brd0, cmd$)
!  Perform board write.
LET wrt(1) = ORD("F") + ORD("3") * 256
LET wrt(2) = ORD("R") + ORD("1") * 256
LET wrt(3) = ORD("X") + ORD("5") * 256
LET wrt(4) = ORD("P") + ORD("2") * 256
LET wrt(5) = ORD("G") + ORD("0") * 256
LET cnt = 10
CALL ibwrti (brd0, wrt(1), cnt)
```

2. To enable automatic byte swapping of binary integer data, see the *IBCONFIG Board Function Example*.

IBWRTIA IBWRTIA

Purpose: Write data asynchronously from integer array.

Format: `CALL ibwrtia (ud, iarr(1), cnt)`

`ud` is a device or an interface board. `iarr` is an array from which integer data is written. `cnt` is the maximum number of bytes to be written.

`ibwrtia` writes asynchronously `cnt` bytes of integer data from `iarr` to the GPIB. The integer data is sent in low-byte, high-byte order to the GPIB.

`ibwrtia` is similar to the `ibwrta` function, which writes data from a character variable.

Refer to *IBWRTA* in this chapter and to *NKR BASIC NI-488 I/O Calls* in Chapter 1.

Device Function Example:

Write ten instruction bytes from integer array `wrt` to the device `dvm` while performing other processing.

```
DECLARE integer wrt(5), cnt
LET wrt(1) = ORD("F") + ORD("3") * 256
LET wrt(2) = ORD("R") + ORD("1") * 256
LET wrt(3) = ORD("X") + ORD("5") * 256
LET wrt(4) = ORD("P") + ORD("2") * 256
LET wrt(5) = ORD("G") + ORD("0") * 256
LET cnt = 10
CALL ibwrtia (dvm, wrt(1), cnt)
LET mask = 16640              !  TIMO CMPL
!  Perform other processing here then wait for I/O
!  completion or a timeout.
CALL ibwait (dvm, mask)
!  ibsta shows how the write terminated: on
!  CMPL, END, TIMO, or ERR.
```

IBWRTIA (continued) IBWRTIA

Board Function Example:

Write ten instruction bytes from the integer array WRT to a device at listen address hex 2F (ASCII /). (The GPIB board talk address is hex 40 or ASCII @.)

```
DECLARE integer wrt(5), cnt
!  Perform addressing.
LET cmd$ = "?@/"              !  UNL MTA LAD
CALL ibcmd (brd0, cmd$)
!  Perform board write.
LET wrt(1) = ORD("F") + ORD("3") * 256
LET wrt(2) = ORD("R") + ORD("1") * 256
LET wrt(3) = ORD("X") + ORD("5") * 256
LET wrt(4) = ORD("P") + ORD("2") * 256
LET wrt(5) = ORD("G") + ORD("0") * 256
LET cnt = 10
CALL ibwrtia (brd0, wrt(1), cnt)
!  Perform other processing here then wait for I/O
!  completion or a timeout.
LET mask = 16640              !  TIMO CMPL
CALL ibwait (brd0, mask)
```

GPIB Programming Examples

These examples illustrate the programming steps that could be used to program a representative IEEE 488 instrument from your personal computer using the NI-488 functions. The applications are written in NKR BASIC. The target instrument is a digital voltmeter (DVM). This instrument is otherwise unspecified (that is, it is not a DVM manufactured by any particular manufacturer). The purpose here is to explain how to use the driver to execute certain programming and control sequences and not how to determine those sequences.

Because the instructions that are sent to program a device as well as the data that might be returned from the device are called *device-dependent messages*, the format and syntax of the messages used in this example are unique to this device. Furthermore, the *interface messages* or bus commands that must be sent to each device will also vary, but to a lesser degree. The exact sequence of messages to program and to control a particular device are contained in its documentation.

For example, the following sequence of actions is assumed to be necessary to program this DVM to make and return measurements of a high frequency AC voltage signal in the autoranging mode:

1. Initialize the GPIB interface circuits of the DVM so that it can respond to messages.

2. Place the DVM in remote programming mode and turn off front panel control.

3. Initialize the internal measurement circuits.

4. Instruct the meter to measure volts alternating current (VAC) using auto-ranging (AUTO), to wait for a trigger from the Controller before starting a measurement (TRIGGER 2), and to assert the IEEE 488 Service Request signal line, SRQ, when the measurement has been completed and the meter is ready to send the result (*SRE 16).

5. For each measurement:

    a. Send the `TRIGGER` command to the multimeter. The `ibwrt` command `"VAL1?"` instructs the meter to send the next triggered reading to its IEEE 488 output buffer.
    b. Wait until the DVM asserts Service Request (SRQ) to indicate that the measurement is ready to be read.

c. Serial poll the DVM to determine if the measured data is valid or if a fault condition exists. You can find out by checking the message available (MAV) bit, bit 4 in the status byte.

d. If the data is valid, read 10 bytes from the DVM.

6. End the session.

The example programs that follow are based on these assumptions:

- The GPIB board is the designated System Active Controller of the GPIB.
- There is no change to the GPIB board default hardware settings.
- The only changes made to the software parameters are those necessary to define the device DVM at primary address 1.
- There is only one GPIB board in use, and it is designated GPIB0.
- The primary listen and talk addresses of GPIB0 are hex 20 (ASCII <space>) and hex 40 (ASCII @), respectively.

NKR BASIC Example Program – Device Functions

```
!  NKR BASIC Example Program - Device Functions

   DECLARE integer ibsta, iberr, ibcnt     !  GPIB status
                                           !  variables.
   DECLARE integer  dvm                    !  Device number.
   DECLARE integer  mask                   !  Wait mask.
   DECLARE integer  spr                    !  Serial poll
                                           !  response.
   DECLARE integer  v                      !  GPIB function
                                           !  parameter.
   DECLARE integer  m                      !  FOR loop index.

!  gpiberr is an error routine that is called when an NI-488
!  function fails.  dvmerr is an error routine that is
!  called when the Fluke 45 does not have valid data to send.

   DECLARE sub gpiberr
   DECLARE sub dvmerr

   CALL cls_1
   PRINT "Read ten measurements from the Fluke 45..."
   PRINT

!  Assign a unique identifier to the Fluke 45 and store it in
!  the variable dvm.  The name "DVM" is the name you
!  configured for the Fluke 45 using IBCONF.EXE.  If dvm is
!  less than zero, call gpiberr with an error message.

   LET devname$ = "DVM"
   CALL ibfind (devname$,dvm)
   IF dvm < 0 then
     LET msg$ = "Ibfind Error"
     CALL gpiberr
     STOP
   END IF

!  Clear the internal or device functions of the Fluke 45.
!  If the error bit (ERR) is set in ibsta, call gpiberr with
!  an error message.

   CALL ibclr (dvm)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "Ibclr Error"
     CALL gpiberr
     STOP
   END IF
```

```
!  Reset the Fluke 45 by issuing the reset (*RST) command.
!  Instruct the Fluke 45 to measure the volts alternating
!  current (VAC) using auto-ranging (AUTO), to wait for a
!  trigger from the GPIB interface board (TRIGGER 2),
!  and to assert the IEEE 488 Service Request line
!  (SRQ) when the measurement has been completed and the
!  Fluke 45 is ready to send the result (*SRE 16).  If
!  the error bit (ERR) is set in ibsta, call gpiberr
!  with an error message.

   LET wrt$ = "*RST; VAC; AUTO; TRIGGER 2; *SRE 16"
   CALL ibwrt (dvm,wrt$)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "Ibwrt Error"
     CALL gpiberr
     STOP
   END IF

!  Initialize the accumulator of the ten measurements to zero.

   LET sum = 0.0

!  Establish a FOR loop to read the ten measurements.  The
!  variable m serves as the counter of the FOR loop.

   FOR m = 1 to 10

      !  Trigger the Fluke 45.  If the error bit (ERR) is set
      !  in ibsta, call gpiberr with an error message.

         CALL ibtrg (dvm)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-32768) <> 0 then
           LET msg$ = "Ibtrg Error"
           CALL gpiberr
           STOP
         END IF

      !  Request the triggered measurement by sending the
      !  instruction "VAL1?".  If the error bit (ERR) is set
      !  in ibsta, call gpiberr with an error message.

         LET wrt$ = "VAL1?"
         call ibwrt(dvm, wrt$)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-32768) <> 0 then
           LET msg$ = "Ibwrt Error"
           CALL gpiberr
           STOP
         END IF
```

```
      !  Wait for the Fluke 45 to request service (RQS) or wait
      !  for the Fluke 45 to timeout (TIMO).  The default
      !  timeout period is 10 seconds.  RQS is detected by bit
      !  position 11 (hex 800).  TIMO is detected by bit
      !  position 14 (hex 4000).  These status bits are listed
      !  under the NI-488 function ibwait in the Software
      !  Reference Manual.  If the error bit (ERR) or the
      !  timeout bit (TIMO) is set in ibsta, call gpiberr with
      !  an error message.

         LET mask = 18432          !  RQS or TIMO
         call ibwait (dvm,mask)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-16384) <> 0 then
           LET msg$ = "Ibwait Error"
           CALL gpiberr
           STOP
         END IF

      !  Read the Fluke 45 serial poll status byte.  If the
      !  error bit (ERR) is set in ibsta, call gpiberr with an
      !  error message.

         call ibrsp (dvm,spr)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-32768) <> 0 then
           LET msg$ = "Ibrsp Error"
           CALL gpiberr
           STOP
         END IF

      !  If the returned status byte is hex 50, the Fluke 45
has
      !  valid data to send; otherwise,it has a fault condition
      !  to report.  If the status byte is not hex 50, call
      !   dvmerr with an error message.

         IF spr <> 80 then
           LET msg$ = "Fluke 45 Error"
           CALL dvmerr
           STOP
         END IF

      !  Read the Fluke 45 measurement.  If the error bit (ERR)
      !  is set in ibsta, call gpiberr with an error message.

         LET rd$ = repeat$(" ",10)
         CALL ibrd (dvm,rd$)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-32768) <> 0 then
           LET msg$ = "Ibrd Error"
           CALL gpiberr
           STOP
         END IF
```

```
      !  Print the measurement received from the Fluke 45.

         LET rd$ = Rd$(1:ibcnt-1)
         PRINT "Reading: "; Rd$
         PRINT

      !  Convert the variable rd to its numeric value and add
      !  to the accumulator.

         LET sum = sum + val(Rd$)

   NEXT m     !  Continue the FOR loop until 10
              !  measurements are read.

!  Print the average of the ten readings.

   PRINT "The average of the 10 readings is ", sum/10

!  Call the ibonl function to disable the hardware
!  and software.

   LET v = 0
   CALL ibonl(dvm, v)

!
================================================================
!                          Procedure gpiberr
!  The gpiberr procedure notifies you that a NI-488 function
!  failed by printing an error message.  The status variable
!  ibsta prints in hexadecimal along with the mnemonic meaning
!  of the bit position.  The status variable iberr prints
!  in decimal along with the mnemonic meaning of the decimal
!  value.  The status variable ibcnt prints in decimal.
!
!  The NI-488 function ibonl is called to disable the hardware
!  and software.
!
================================================================
SUB gpiberr
   PRINT msg$

   PRINT "ibsta = &H"; hex$(ibsta_i)

   IF b_AND(ibsta_i, -32768)    <>    0 then PRINT " ERR"
   IF b_AND(ibsta_i, 16384)     <>    0 then PRINT " TIMO"
   IF b_AND(ibsta_i, 8192)      <>    0 then PRINT " END"
   IF b_AND(ibsta_i, 4096)      <>    0 then PRINT " SRQI"
   IF b_AND(ibsta_i, 2048)      <>    0 then PRINT " RQS"
   IF b_AND(ibsta_i, 256)       <>    0 then PRINT " CMPL"
   IF b_AND(ibsta_i, 128)       <>    0 then PRINT " LOK"
   IF b_AND(ibsta_i, 64)        <>    0 then PRINT " REM"
   IF b_AND(ibsta_i, 32)        <>    0 then PRINT " CIC"
   IF b_AND(ibsta_i, 16)        <>    0 then PRINT " ATN"
   IF b_AND(ibsta_i, 8)         <>    0 then PRINT " TACS"
   IF b_AND(ibsta_i, 4)         <>    0 then PRINT " LACS"
   IF b_AND(ibsta_i, 2)         <>    0 then PRINT " DTAS"
```

```
   IF b_AND(ibsta_i, 1)         <>    0 then PRINT " DCAS"
   PRINT

   PRINT "iberr = ", iberr
   IF iberr =  0 then PRINT " EDVR <DOS Error>"
   IF iberr =  1 then PRINT " ECIC <Not CIC>"
   IF iberr =  2 then PRINT " ENOL <No Listener>"
   IF iberr =  3 then PRINT " EADR <Address error>"
   IF iberr =  4 then PRINT " EARG <Invalid argument>"
   IF iberr =  5 then PRINT " ESAC <Not Sys Ctrlr>"
   IF iberr =  6 then PRINT " EABO <Op. aborted>"
   IF iberr =  7 then PRINT " ENEB <No GPIB board>"
   IF iberr = 10 then PRINT " EOIP <Async I/O in prg>"
   IF iberr = 11 then PRINT " ECAP <No capability>"
   IF iberr = 12 then PRINT " EFSO <File sys. error>"
   IF iberr = 14 then PRINT " EBUS <Command error>"
   IF iberr = 15 then PRINT " ESTB <Status byte lost>"
   IF iberr = 16 then PRINT " ESRQ <SRQ stuck on>"
   IF iberr = 20 then PRINT " ETAB <Table Overflow>"
   PRINT

   PRINT "ibcnt = ", ibcnt

!  Call the ibonl function to disable the hardware and
!  software.

   LET v = 0
   CALL ibonl(dvm, v)

END SUB

!
!==============================================================
!                          Procedure dvmerr
!  The dvmerr procedure notifies you that the Fluke 45 returned
!  an invalid serial poll response byte.  The error message
!  prints along with the serial poll response byte.
!
!  The NI-488 function ibonl is called to disable the hardware
!  and software.
!
!==============================================================
SUB dvmerr

   PRINT msg$
   LET spr_i = spr
   PRINT "Returned Byte = &H"; hex$(spr_i)
```

```
!  Call the ibonl function to disable the hardware.

   LET v = 0
   CALL ibonl (dvm, v)

END SUB

END
```

NKR BASIC Example Program – Board Functions

```
!  NKR BASIC Example Program - Board Functions

   DECLARE integer ibsta, iberr, ibcnt     !  GPIB status
                                           !  variables.
   DECLARE integer  brd0                   !  Board number.
   DECLARE integer  mask                   !  Wait mask.
   DECLARE integer  v                      !  GPIB function
                                           !  parameter.
   DECLARE integer  m                      !  FOR loop index.

!  gpiberr is an error routine that is called when an NI-488
!  function fails.  dvmerr is an error routine that is
!  called when the Fluke 45 does not have valid data the send.

   DECLARE sub gpiberr
   DECLARE sub dvmerr

   CALL cls_1
   PRINT "Read ten measurements from the Fluke 45..."
   PRINT

!  Assign a unique identifier to board 0 and store it in the
!  variable BRD0.  The name 'GPIB0' is the default name of
!  board 0.  If BRD0 is less than zero, call gpiberr with
!  an error message.

   LET bdname$ = "GPIB0"
   CALL ibfind(bdname$, brd0)
   IF brd0 < 0 then
     LET msg$ = "Ibfind Error"
     CALL gpiberr
     STOP
   END IF

!  Send the Interface Clear (IFC) message.  This action
!  initializes the GPIB interface board and makes the
!  interface board Controller-In-Charge.  If the error bit
!  (ERR) is set in ibsta, call gpiberr with an error message.

   CALL ibsic(brd0)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "Ibsic Error"
     CALL gpiberr
     STOP
   END IF

!  Turn on the Remote Enable (REN) signal.  The device does
!  not actually enter remote mode until it receives its
!  listen address.  If the error bit (ERR) is set in ibsta,
!  call gpiberr with an error message.
```

```
   LET v = 1
   CALL ibsre(brd0, v)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "Ibsre Error"
     CALL gpiberr
     STOP
   END IF

!  Inhibit front panel control with the Local Lockout (LLO)
!  command (hex 11).  Place the Fluke 45 in remote mode
!  by addressing it to listen (hex 21 or ASCII "!").
!  Send the Device Clear (DCL) message to clear internal
!  device functions (hex 14).  Address the GPIB interface
!  board to talk (hex 40 or ASCII "@").  These commands
!  can be found in Appendix A of the Software Reference
!  Manual.  If the error bit (ERR) is set in ibsta, call
!  gpiberr with an error message.

   LET cmd$ = chr$(17) & "!" & chr$(20) & "@"
   CALL ibcmd(brd0, cmd$)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "Ibcmd Error"
     CALL gpiberr
     STOP
   END IF

!  Reset the Fluke 45 by issuing the reset (*RST) command.
!  Instruct the Fluke 45 to measure the volts alternating
!  current (VAC) using auto-ranging (AUTO), to wait for a
!  trigger from the GPIB interface board (TRIGGER 2),
!  and to assert the IEEE 488 Service Request line (SRQ)
!  when the measurement has been completed and the Fluke 45
!  is ready to send the result (*SRE 16).  If the error
!  bit (ERR) is set in ibsta, call gpiberr with an error
!  message.

   LET wrt$ = "*RST; VAC; AUTO; TRIGGER 2; *SRE 16"
   CALL ibwrt(brd0, wrt$)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "Ibwrt Error"
     CALL gpiberr
     STOP
   END IF

!  Initialize the accumulator of the ten measurements
!  to zero.

   LET sum = 0.0
```

```
!  Establish a FOR loop to read the ten measurements.  The
!  variable m serves as the counter of the FOR loop.

   FOR m = 1 to 10

      !  Address the Fluke 45 to listen (hex 21 or ASCII "!")
      !  and address the GPIB interface board to talk
      !  (hex 48 or ASCII "@").  These commands can be found
      !  in Appendix A of the Software Reference Manual.  If
      !  the error bit (ERR) is set in ibsta, call gpiberr with
      !  an error message.

         LET cmd$ = "!" & "@"
         CALL ibcmd(brd0, cmd$)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-32768) <> 0 then
           LET msg$ = "Ibcmd Error"
           CALL gpiberr
           STOP
         END IF

      !  Trigger the Fluke by sending the trigger (GET) command
      !  (hex 08) message.  If the error bit (ERR) is set in
      !  ibsta, call gpiberr with an error message.

         LET cmd$ = chr$(8)
         CALL ibcmd(brd0, cmd$)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-32768) <> 0 then
           LET msg$ = "Ibcmd Error"
           CALL gpiberr
           STOP
         END IF

      !  Request the triggered measurement by sending the
      !  instruction "VAL1?".  If the error bit (ERR) is set
      !  ibsta, call gpiberr with an error message.

         LET wrt$ = "VAL1?"
         CALL ibwrt(brd0, wrt$)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-32768) <> 0 then
           LET msg$ = "Ibwrt Error"
           CALL gpiberr
           STOP
         END IF
```

```
      !  Wait for the Fluke 45 to assert the Service Request
      !  (SRQ) line or wait for the Fluke 45 to timeout(TIMO).
      !  The default timeout period is 10 seconds.  SRQ is
      !  detected by bit position 12 (hex 1000, SRQI).  TIMO
      !  is detected by bit position 14 (hex 4000).  These
      !  status bits are listed under the NI-488 function
      !  ibwait in the Software Reference Manual.  If error
      !  bit (ERR) or the timeout bit (TIMO) is set in ibsta,
      !  call gpiberr with an error message.

         LET mask = 20480         !  SRQI or TIMO
         CALL ibwait(brd0, mask)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-16384) <> 0 then
           LET msg$ = "Ibwait Error"
           CALL gpiberr
           STOP
         END IF

      !  Serial poll the Fluke 45.  Unaddress bus devices by
      !  sending the untalk (UNT) command (hex 5F or ASCII "_")
      !  and the unlisten (UNL) command (hex 3F or ASCII "?").
      !  Send the Serial Poll Enable (SPE) command (hex 18) and
      !  the Fluke 45 talk address (hex 41 or ASCII "A").
      !  Address the GPIB interface board to listen (hex 20 or
      !  ASCII space).  These commands can be found in
      !  Appendix A of the Software Reference Manual.  If the
      !  error bit (ERR) is set in ibsta, call gpiberr with
      !  an error message.

         LET cmd$ = "_?" & chr$(24) & "A "
         CALL ibcmd(brd0, cmd$)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-32768) <> 0 then
           LET msg$ = "Ibcmd Error"
           CALL gpiberr
           STOP
         END IF

      !  Read the Fluke 45 serial poll status byte. If the
error
      !  bit (ERR) is set in ibsta, call gpiberr with an
      !  error message.

         LET rd$ = repeat$(" ",1)
         CALL ibrd(brd0, rd$)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-32768) <> 0 then
           LET msg$ = "Ibrd Error"
           CALL gpiberr
           STOP
         END IF
```

```
      !  If the returned status byte is hex 50, the Fluke 45
      !  has valid data to send; otherwise, it has a fault
      !  condition to report.  If the status byte is not
      !  hex 50, call dvmerr with an error message.

         IF ord(rd$) <> 80 then
           LET msg$ = "Fluke 45 Error"
           CALL dvmerr
           STOP
         END IF

      !  Complete the serial poll by sending the Serial Poll
      !  Disable (SPD) command, hex 19.  This command can be
      !  found in Appendix A of the Software Reference
      !  Manual.  If the error bit (ERR) is set in ibsta,
      !  call gpiberr with an error message.

         LET cmd$ = chr$(25)
         CALL ibcmd(brd0, cmd$)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-32768) <> 0 then
           LET msg$ = "Ibcmd Error"
           CALL gpiberr
           STOP
         END IF

      !  Read the Fluke 45 measurement.  If the error bit (ERR)
      !  is set in ibsta, call gpiberr with an error message.

         LET rd$ = repeat$(" ",10)
         CALL ibrd(brd0, rd$)
         LET ibsta_i = ibsta
         IF b_AND(ibsta_i,-32768) <> 0 then
           LET msg$ = "IBRD Error"
           CALL gpiberr
           STOP
         END IF

      !  Print the measurement received from the Fluke 45.

         LET rd$ = Rd$(1:ibcnt-1)
         PRINT "Reading: "; Rd$
         PRINT

      !  Convert the variable rd to its numeric value and add
      !  to the accumulator.

         LET sum = sum + val(Rd$)

   NEXT m     !  Continue the FOR loop until 10 measurements
              !  are read.
```

```
!  Print the average of the ten readings.

   PRINT "The average of the 10 readings is ", sum/10

!  Call the ibonl function to disable the hardware
!  and software.

   LET v = 0
   CALL ibonl(brd0, v)


! ============================================================
!                         Procedure gpiberr
!  The gpiberr procedure notifies you that a NI-488 function
!  failed by printing an error message.  The status variable
!  ibsta prints in hexadecimal along with the mnemonic meaning
!  of the bit position.The status variable iberr prints in
!  decimal along with the mnemonic meaning of the decimal
!  value.  The status variable ibcnt prints in decimal.
!
!  The NI-488 function ibonl is called to disable the hardware
!  and software.
! ============================================================
SUB gpiberr

   PRINT msg$

   PRINT "ibsta = &H"; hex$(ibsta_i)

   IF b_AND(ibsta_i,-32768)    <>  0 then PRINT " ERR"
   IF b_AND(ibsta_i,16384)     <>  0 then PRINT " TIMO"
   IF b_AND(ibsta_i,8192)      <>  0 then PRINT " END"
   IF b_AND(ibsta_i,4096)      <>  0 then PRINT " SRQI"
   IF b_AND(ibsta_i,2048)      <>  0 then PRINT " RQS"
   IF b_AND(ibsta_i,256)       <>  0 then PRINT " CMPL"
   IF b_AND(ibsta_i,128)       <>  0 then PRINT " LOK"
   IF b_AND(ibsta_i,64)        <>  0 then PRINT " REM"
   IF b_AND(ibsta_i,32)        <>  0 then PRINT " CIC"
   IF b_AND(ibsta_i,16)        <>  0 then PRINT " ATN"
   IF b_AND(ibsta_i,8)         <>  0 then PRINT " TACS"
   IF b_AND(ibsta_i,4)         <>  0 then PRINT " LACS"
   IF b_AND(ibsta_i,2)         <>  0 then PRINT " DTAS"
   IF b_AND(ibsta_i,1)         <>  0 then PRINT " DCAS"
   PRINT

   PRINT "iberr = ", iberr
   IF iberr =  0 then PRINT " EDVR <DOS Error>"
   IF iberr =  1 then PRINT " ECIC <Not CIC>"
   IF iberr =  2 then PRINT " ENOL <No Listener>"
   IF iberr =  3 then PRINT " EADR <Address error>"
   IF iberr =  4 then PRINT " EARG <Invalid argument>"
   IF iberr =  5 then PRINT " ESAC <Not Sys Ctrlr>"
   IF iberr =  6 then PRINT " EABO <Op. aborted>"
   IF iberr =  7 then PRINT " ENEB <No GPIB board>"
   IF iberr = 10 then PRINT " EOIP <Async I/O in prg>"
   IF iberr = 11 then PRINT " ECAP <No capability>"
```

```
   IF iberr = 12 then PRINT " EFSO <File sys. error>"
   IF iberr = 14 then PRINT " EBUS <Command error>"
   IF iberr = 15 then PRINT " ESTB <Status byte lost>"
   IF iberr = 16 then PRINT " ESRQ <SRQ stuck on>"
   IF iberr = 20 then PRINT " ETAB <Table Overflow>"
   PRINT

   PRINT "ibcnt = ", ibcnt

!  Call the ibonl function to disable the hardware and
!  software.

   LET v = 0
   CALL ibonl(brd0, v)

END SUB

!
================================================================
!                          Routine dvmerr
!  The dvmerr routine notifies you that the Fluke 45 returned
!  an invalid serial poll response byte.  The error message
!  prints along with the serial poll response byte.
!
!  The NI-488 function ibonl is called to disable the hardware
!  and software.
!
================================================================
SUB dvmerr

   PRINT msg$

   PRINT "Status Byte = ", ord(rd$)

!  Call the ibonl function to disable the hardware and
!  software.

   LET v = 0
   CALL ibonl(brd0, v)

END SUB

END
```

Appendix A
Multiline Interface Messages

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

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

Multiline Interface Messages

Hex	Oct	Dec	ASCII	Msg	Hex	Oct	Dec	ASCII	Msg
00	000	0	NUL		20	040	32	SP	MLA0
01	001	1	SOH	GTL	21	041	33	!	MLA1
02	002	2	STX		22	042	34	"	MLA2
03	003	3	ETX		23	043	35	#	MLA3
04	004	4	EOT	SDC	24	044	36	$	MLA4
05	005	5	ENQ	PPC	25	045	37	%	MLA5
06	006	6	ACK		26	046	38	&	MLA6
07	007	7	BEL		27	047	39	'	MLA7
08	010	8	BS	GET	28	050	40	(	MLA8
09	011	9	HT	TCT	29	051	41	)	MLA9
0A	012	10	LF		2A	052	42	*	MLA10
0B	013	11	VT		2B	053	43	+	MLA11
0C	014	12	FF		2C	054	44	,	MLA12
0D	015	13	CR		2D	055	45	-	MLA13
0E	016	14	SO		2E	056	46	.	MLA14
0F	017	15	SI		2F	057	47	/	MLA15
10	020	16	DLE		30	060	48	0	MLA16
11	021	17	DC1	LLO	31	061	49	1	MLA17
12	022	18	DC2		32	062	50	2	MLA18
13	023	19	DC3		33	063	51	3	MLA19
14	024	20	DC4	DCL	34	064	52	4	MLA20
15	025	21	NAK	PPU	35	065	53	5	MLA21
16	026	22	SYN		36	066	54	6	MLA22
17	027	23	ETB		37	067	55	7	MLA23
18	030	24	CAN	SPE	38	070	56	8	MLA24
19	031	25	EM	SPD	39	071	57	9	MLA25
1A	032	26	SUB		3A	072	58	:	MLA26
1B	033	27	ESC		3B	073	59	;	MLA27
1C	034	28	FS		3C	074	60	<	MLA28
1D	035	29	GS		3D	075	61	=	MLA29
1E	036	30	RS		3E	076	62	>	MLA30
1F	037	31	US		3F	077	63	?	UNL

Message Definitions

DCL	Device Clear	MSA	My Secondary Address
GET	Group Execute Trigger	MTA	My Talk Address
GTL	Go To Local	PPC	Parallel Poll Configure
LLO	Local Lockout	PPD	Parallel Poll Disable
MLA	My Listen Address		

Multiline Interface Messages

Hex	Oct	Dec	ASCII	Msg	Hex	Oct	Dec	ASCII	Msg
40	100	64	@	MTA0	60	140	96	`	MSA0,PPE
41	101	65	A	MTA1	61	141	97	a	MSA1,PPE
42	102	66	B	MTA2	62	142	98	b	MSA2,PPE
43	103	67	C	MTA3	63	143	99	c	MSA3,PPE
44	104	68	D	MTA4	64	144	100	d	MSA4,PPE
45	105	69	E	MTA5	65	145	101	e	MSA5,PPE
46	106	70	F	MTA6	66	146	102	f	MSA6,PPE
47	107	71	G	MTA7	67	147	103	g	MSA7,PPE
48	110	72	H	MTA8	68	150	104	h	MSA8,PPE
49	111	73	I	MTA9	69	151	105	i	MSA9,PPE
4A	112	74	J	MTA10	6A	152	106	j	MSA10,PPE
4B	113	75	K	MTA11	6B	153	107	k	MSA11,PPE
4C	114	76	L	MTA12	6C	154	108	l	MSA12,PPE
4D	115	77	M	MTA13	6D	155	109	m	MSA13,PPE
4E	116	78	N	MTA14	6E	156	110	n	MSA14,PPE
4F	117	79	O	MTA15	6F	157	111	o	MSA15,PPE
50	120	80	P	MTA16	70	160	112	p	MSA16,PPD
51	121	81	Q	MTA17	71	161	113	q	MSA17,PPD
52	122	82	R	MTA18	72	162	114	r	MSA18,PPD
53	123	83	S	MTA19	73	163	115	s	MSA19,PPD
54	124	84	T	MTA20	74	164	116	t	MSA20,PPD
55	125	85	U	MTA21	75	165	117	u	MSA21,PPD
56	126	86	V	MTA22	76	166	118	v	MSA22,PPD
57	127	87	W	MTA23	77	167	119	w	MSA23,PPD
58	130	88	X	MTA24	78	170	120	x	MSA24,PPD
59	131	89	Y	MTA25	79	171	121	y	MSA25,PPD
5A	132	90	Z	MTA26	7A	172	122	z	MSA26,PPD
5B	133	91	[	MTA27	7B	173	123	{	MSA27,PPD
5C	134	92	\	MTA28	7C	174	124	|	MSA28,PPD
5D	135	93	]	MTA29	7D	175	125	}	MSA29,PPD
5E	136	94	^	MTA30	7E	176	126	~	MSA30,PPD
5F	137	95	_	UNT	7F	177	127	DEL	

PPE	Parallel Poll Enable	SPE	Serial Poll Enable
PPU	Parallel Poll Unconfigure	TCT	Take Control
SDC	Selected Device Clear	UNL	Unlisten
SPD	Serial Poll Disable	UNT	Untalk

Appendix B
Applications Monitor

This appendix introduces you to the Applications Monitor, a resident program that is useful in debugging sequences of GPIB calls from within your application.

The applications monitor can temporarily halt program execution (trap) upon return from NI-488 functions that meet a condition specified by you. You then can inspect function arguments, buffers, return values, GPIB global variables, and other pertinent data. You can select the condition that halts the program on every NI-488 function, on those functions that return an error indication, or on those calls that are returned with selected bit patterns in the GPIB status word.

If the desired condition is met, you will see a pop-up screen (Figure B-1) that contains details of the call being trapped. In addition, you can view up to 255 of the preceding calls to verify that the sequence of calls and their arguments have occurred as intended.

Figure B-1. Applications Monitor Pop-Up Screen

In many cases, you can omit explicit error-checking if you use the applications monitor. If a program is expected to run without errors, trapping on errors will cause the applications monitor to be invoked only if an error occurs during a GPIB call. You can then take the action necessary to fix the problem.

Currently, the applications monitor is available with all NI-488.2 MS-DOS drivers.

Installing the Applications Monitor

The applications monitor is included on the distribution diskette as the file `APPMON.EXE`. To install it, type the following command in response to the DOS prompt:

```
APPMON
```

If the GPIB driver is not present or the applications monitor has been previously installed, it will not load and an error message will be printed.

Once installed, the applications monitor will remain in memory until you restart the system. Should you later decide that you no longer wish to devote memory to the resident applications monitor, simply restart your system; the applications monitor will no longer be loaded.

IBTRAP

Once installed, the applications monitor is run by the `ibtrap` function. The applications monitor can trap on GPIB driver calls that have certain bits set in the GPIB status word. The trap options are set by the special GPIB driver call, `ibtrap`. This call can be made either from the application program, or from DOS prompt using the special utility program called `IBTRAP.EXE`.

With the function call and the DOS utility you select a *mask*, which determines those events that will be trapped, and a *monitor mode*, which selects what is displayed when a trap occurs.

The exact syntax of the function call is dependent on the language you are using. See the description of `ibtrap` in your language section for details about how to include `ibtrap` calls in your application.

The utility program `ibtrap` can be used to set the trap mode from DOS. Simply type `ibtrap` in response to the DOS prompt, specifying the desired combination of the flags listed on the following pages.

Select one or more mask flags:

`-ALL`	all GPIB calls
`-ERR`	GPIB error
`-TIMO`	timeout
`-END`	GPIB board detected END or EOS
`-SRQI`	SRQ on
`-RQS`	device requesting service
`-CMPL`	I/O completed
`-LOK`	GPIB board is in Lockout State
`-REM`	GPIB board is in Remote State
`-CIC`	GPIB board is Controller-In-Charge
`-ATN`	attention is asserted
`-TACS`	GPIB board is Talker
`-LACS`	GPIB board is Listener
`-DTAS`	GPIB board is in Device Trigger State
`-DCAS`	GPIB board is in Device Clear State

Select only one monitor flag:

`-OFF` turns the monitor off. No recording or trapping occurs.

`-REC` instructs the monitor to record all GPIB driver calls but no trapping occurs.

`-DIS` instructs the monitor to record all GPIB driver calls and display whenever a trap condition exists.

Omitting either the mask or the monitor flags will leave its current configuration unchanged. Invoking `ibtrap` without any flags will display the valid flags and their current state. This has no effect on the applications monitor configuration.

By selecting various flags for the mask and monitor parameters, you can achieve a variety of trapping configurations. The following are some examples:

`IBTRAP -CIC -ATN -DIS`	record all GPIB driver calls and display the applications monitor whenever attention is asserted or the GPIB board is Controller-in-Charge.
`IBTRAP -SRQ -REC`	record all GPIB driver calls and set the trap mask to trap when SRQ is on. Do not display the applications monitor when a trap condition exists.
`IBTRAP -DIS`	record all GPIB driver calls and display the applications monitor whenever a trap condition exists. The trap mask remains unchanged.
`IBTRAP -OFF`	disable the applications monitor. No recording or trapping is performed.

See Chapter 3 of this manual for the appropriate syntax to use in your application program.

Applications Monitor Options

When the applications monitor is displayed, you can view the parameters of the current GPIB call, change the display and trap modes, and scan the GPIB session summary. The applications monitor displays the following information pertinent to the current GPIB call:

- `Device` symbolic device name.
- `Function` NI-488 call or function mnemonic and description.
- `Value` for functions that have a number as their second parameter, this contains its value; otherwise, it is undefined.
- `Count` for functions that have a count as their third parameter, this contains its value; otherwise, it is undefined.
- `ibsta` contains the GPIB status information.
- `iberr` contains the GPIB error information or the previous value of the `value` parameter if no error occurred.
- `ibcnt` contains the number of bytes transferred.
- `Buffer Value` for functions that have a buffer as a parameter, this displays its contents. Each byte of the buffer is shown with its index, character image, and ASCII value.
- `Status` shows the state of the individual bits of `ibsta`. A "*" indicates the bit is active. The active bits of the trap mask are highlighted for easy identification.
- `Error` shows the state of the individual bits of `iberr`. A "*" indicates the bit is active.
- `Information` contains any message concerning the current GPIB call.

Note: All numbers are displayed in hex. Also, the applications monitor is unable to record `ibfind` or `ibtrap` calls.

Main Commands

When the main applications monitor screen is displayed, the following command keys are available:

<F1>	continue executing applications program
<F2>	display session summary
<F3>	exit to DOS
<F5>	configure trap mask
<F6>	configure monitor mode
<F7>	hide/show monitor
<F8>	clear session summary buffer
<F10>	display command key list
<Cursor Up>	scroll buffer up one character
<Cursor Down>	scroll buffer down one character
<Page Up>	scroll buffer up one page
<Page Down>	scroll buffer down one page
<Home>	scroll to beginning of buffer
<End>	scroll to end of buffer

Session Summary Screen

This session summary can be viewed by pressing F2. Once displayed, the following keys can manipulate the display:

<Cursor Up>	scrolls summary up one line
<Cursor Down>	scrolls summary down one line
<Page Up>	scrolls summary up one page
<Page Down>	scrolls summary down one page
<Home>	scrolls to the top of summary
<End>	scrolls to the end of summary
<Escape > or <F2>	exits the session summary display and returns to the main applications monitor screen

Configuring the Trap Mask

Press <F5> to change the current configuration of the trap mask. This action yields a popup menu with each of the status bits displayed along with their current state (either ON or OFF). Press the up and down arrow keys to highlight the desired bit and press <F1> to toggle its state. Press <Enter> to record the changes. Pressing <Escape> cancels this action and leaves the mask unchanged. Selecting all bits has the effect of trapping on every call, while turning them all off causes no trapping to occur.

Configuring the Monitor Mode

Press <F6> to change the current configuration of the applications monitor mode. This action yields a popup menu with the current mode checkmarked. Use the up and down arrow keys to highlight the new mode and press <Enter> to record the change. Press <Escape> to cancel this action and leave the mode unchanged.

Hiding and Showing the Applications Monitor

Press <F7> to hide the applications monitor and restore the contents of the screen. By pressing <F7>, you can view program output written to the screen, within the applications monitor, while the program is active. Pressing <F7> again will restore the applications monitor.

Exiting Directly to DOS

Press <F3> to exit directly from your application back to DOS. This will terminate your application and let you continue working from the DOS prompt.

Appendix C
Customer Communication

For your convenience this appendix contains forms to help you gather the information necessary to help us solve possible technical problems, as well as a form you can use to comment on the product documentation.

By completing these forms before calling National Instruments, you will save yourself time, and our applications engineers will be able to answer your questions more accurately and efficiently. The forms contain the information that the applications engineers need from you to help solve your problem. Briefly jot down the information requested on the line after each item.

Fax Technical Support

If you encounter any technical problems, please complete the fax and configuration forms before requesting technical support by fax. You can contact us by fax at any time at the following number:

(512) 794-5678

Telephone Technical Support

For best service by telephone, please complete the fax and configuration forms, record any error messages, and be available at your computer when you call for technical support. You can use the following numbers between the hours of 8:00 a.m. and 5:30 p.m. (central time) to call the National Instruments applications engineering department:

(512) 794-0100
(800) IEEE-488 (toll-free U.S. and Canada)

Documentation Comments

You can use the *Documentation Comment Form* for your comments about our documents. Please mail or fax it to National Instruments.

Technical Support Fax Form

Technical support is available at any time by fax at (512) 794-5678. For best results, provide as much information as possible. Include the information from your configuration form. Use additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___)_____ Phone (___)_____

Computer brand _____

Model _____ Processor _____

Operating system _____

Speed _____MHz RAM _____M

Display adapter _____

Mouse _____yes _____no

Other adapters installed_____

Hard disk capacity _____M 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 _____

_____

_____

_____

_____

_____

_____

NKR BASIC Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Update this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration.

National Instruments Products

- NI-488.2 Software Revision Number on Disk _____
- Programming Language Interface Revision _____
- Type of National Instruments GPIB boards installed and their respective hardware settings:

Board Type	Interrupt Level	DMA Channel	Base I/O Address

(continues)

Other Products

- Computer Make and Model _____
- Microprocessor _____
- Clock Frequency _____
- Type of Monitor Card Installed _____
- DOS Version _____
- Programming Language/Version _____
- Type of other boards installed and their respective hardware settings:

Board Type	Base I/O Address	Interrupt Level	DMA Channel
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____

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 and NI-488.2 Subroutines for NKR BASIC**

Edition Date: **August 1992**

Part Number: **320348-01**

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

```
! NKR BASIC Example Program - Board Functions

  DECLARE integer  ibsta, iberr, ibcnt        ! GPIB status variables
  DECLARE integer  brd0                       ! Board number
  DECLARE integer  mask                       ! Wait mask
  DECLARE integer  v                          ! GPIB function parameter
  DECLARE integer  m                          ! FOR loop index

! GPIBERR is an error subroutine that is called when a NI-488 function fails.
! DVMERR is an error subroutine that is called when the Fluke 45 does not
! have valid data to send.

  DECLARE sub gpiberr
  DECLARE sub dvmerr

  CALL cls_1
  PRINT "Read ten measurements from the Fluke 45..."
  PRINT

! Assign a unique identifier to board 0 and store in the variable BRD0.
! The name 'GPIB0' is the default name of board 0.  If BRD0 is less
! than zero, call GPIBERR with an error message.

  LET bdname$ = "GPIB0"
  CALL ibfind(bdname$, brd0)
  IF brd0 < 0 then
   LET msg$ = "Ibfind Error"
   CALL gpiberr
   STOP
  END IF

! Send the Interface Clear (IFC) message.  This action initializes the
! GPIB interface board and makes the interface board Controller-In-Charge.
! If the error bit ERR is set in IBSTA, call GPIBERR with an error
! message.

  CALL ibsic(brd0)
  LET ibsta_i = ibsta
  IF b_AND(ibsta_i,-32768) <> 0 then
   LET msg$ = "Ibsic Error"
   CALL gpiberr
   STOP
  END IF

! Turn on the Remote Enable (REN) signal.  The device does not actually
! enter remote mode until it receives its listen address.  If the
! error bit ERR is set in IBSTA, call GPIBERR with an error message.

  LET v = 1
```

```
! Inhibit front panel control with the Local Lockout (LLO) command
! (hex 11).  Place the Fluke 45 in remote mode by addressing it to listen
! (hex 21 or ASCII "!").  Send the Device Clear (DCL) message to clear
! internal device functions (hex 14).  Address the GPIB interface board to
! talk (hex 20 or ASCII "@").  These commands can be found in Appendix A of
! the Software Reference Manual.  If the error bit ERR is set in IBSTA,
! call GPIBERR with an error message.

  LET cmd$ = chr$(17) & "!" & chr$(20) & "@"
  CALL ibcmd(brd0, cmd$)
  LET ibsta_i = ibsta
  IF b_AND(ibsta_i,-32768) <> 0 then
    LET msg$ = "Ibcmd Error"
    CALL gpiberr
    STOP
  END IF

! Reset the Fluke 45 by issuing the reset (*RST) command.  Instruct the
! Fluke 45 to measure the volts alternating current (VAC) using auto-ranging
! (AUTO), to wait for a trigger from the GPIB interface board (TRIGGER 2),
! and to assert the IEEE-488 Service Request line, SRQ, when the measurement
! has been completed and the Fluke 45 is ready to send the result (*SRE 16).
! If the error bit ERR is set in IBSTA, call GPIBERR with an error message.

  LET wrt$ = "*RST; VAC; AUTO; TRIGGER 2; *SRE 16"
  CALL ibwrt(brd0, wrt$)
  LET ibsta_i = ibsta
  IF b_AND(ibsta_i,-32768) <> 0 then
    LET msg$ = "Ibwrt Error"
    CALL gpiberr
    STOP
  END IF

! Initialize the accumulator of the ten measurements to zero.

  LET sum = 0.0

! Establish FOR loop to read the ten measuements.  The variable m will
! serve as the counter of the FOR loop.

  FOR m = 1 to 10

    ! Address the Fluke 45 to listen (hex 21 or ASCII "!") and address the
    ! GPIB interface board to talk (hex 20 or ASCII "@").  These commands
    ! can be found in Appendix A of the Software Reference Manual.  If
    ! the error bit ERR is set in IBSTA, call GPIBERR with an error
    ! message.

      LET cmd$ = "!" & "@"
```

```
! Trigger the Fluke by sending the trigger (GET) command (hex 08)
! message. If the error bit ERR is set in IBSTA, call GPIBERR
! with an error message.

  LET cmd$ = chr$(8)
  CALL ibcmd(brd0, cmd$)
  LET ibsta_i = ibsta
  IF b_AND(ibsta_i,-32768) <> 0 then
   LET msg$ = "Ibcmd Error"
   CALL gpiberr
   STOP
  END IF

! Request the triggered measurement by sending the instruction "VAL1?".
! If the error bit ERR is set IBSTA, call GPIBERR with an error
! message.

  LET wrt$ = "VAL1?"
  CALL ibwrt(brd0, wrt$)
  LET ibsta_i = ibsta
  IF b_AND(ibsta_i,-32768) <> 0 then
   LET msg$ = "Ibwrt Error"
   CALL gpiberr
   STOP
  END IF

! Wait for the Fluke 45 to assert the Service Request (SRQ) line
! or wait for the Fluke 45 to timeout(TIMO). The default timeout
! period is 10 seconds. SRQ is detected by bit position 12
! (hex 1000, SRQI). TIMO is detected by bit position 14 (hex 4000).
! These status bits are listed under the NI-488 function IBWAIT in
! the Software Reference Manual. If error bit ERR or the timeout
! bit TIMO is set in IBSTA, call GPIBERR with an error message.

  LET mask = 20480                ! SRQI or TIMO
  CALL ibwait(brd0, mask)
  LET ibsta_i = ibsta
  IF b_AND(ibsta_i,-16384) <> 0 then
   LET msg$ = "Ibwait Error"
   CALL gpiberr
   STOP
  END IF

! Serial poll the Fluke 45. Unaddress bus devices by sending the
! untalk (UNT) command (hex 5F or ASCII "_") and the unlisten (UNL)
! command (hex 3F or ASCII "?"). Send the Serial Poll Enable (SPE)
! command (hex 18) and the Fluke 45 talk address (hex 41 or ASCII "A").
! Address the GPIB interface board to listen (hex 20 or ASCII space).
! These commands can be found in Appendix A of the Software Reference
! Manual. If the error bit ERR is set in IBSTA, call GPIBERR
! with an error message.
```

```
! Read the Fluke 45 serial poll status byte.  If the error bit
! ERR is set in IBSTA, call GPIBERR with an error message.

  LET rd$ = repeat$(" ",1)
  CALL ibrd(brd0, rd$)
  LET ibsta_i = ibsta
  IF b_AND(ibsta_i,-32768) <> 0 then
   LET msg$ = "Ibrd Error"
   CALL gpiberr
   STOP
  END IF

! If the returned status byte is hex 50, the Fluke 45 has valid data to
! send; otherwise, it has a fault condition to report.  If the status
! byte is not hex 50, call DVMERR with an error message.

  IF ord(rd$) <> 80 then
   LET msg$ = "Fluke 45 Error"
   CALL dvmerr
   STOP
  END IF

! Complete the serial poll by sending the Serial Poll Disable (SPD)
! command, hex 19.  This command can be found in Appendix A of the
! Software Reference Manual.  If the error bit ERR is set in IBSTA,
! call GPIBERR with an error message.

  LET cmd$ = chr$(25)
  CALL ibcmd(brd0, cmd$)
  LET ibsta_i = ibsta
  IF b_AND(ibsta_i,-32768) <> 0 then
   LET msg$ = "Ibcmd Error"
   CALL gpiberr
   STOP
  END IF

! Read the Fluke 45 measurement.  If the error bit ERR is set in
! IBSTA, call GPIBERR with an error message.

  LET rd$ = repeat$(" ",10)
  CALL ibrd(brd0, rd$)
  LET ibsta_i = ibsta
  IF b_AND(ibsta_i,-32768) <> 0 then
   LET msg$ = "IBRD Error"
   CALL gpiberr
   STOP
  END IF

! Print the measurement received from the Fluke 45.
```

```
! Print the average of the ten readings.

  PRINT "The average of the 10 readings is ", sum/10

! Call the ibonl function to disable the hardware and software.

  LET v = 0
  CALL ibonl(brd0, v)


! ============================================================
!                    Subroutine GPIBERR
! This subroutine will notify you that a NI-488.2 function failed by
! printing an error message.  The status variable IBSTA will also be
! printed in hexadecimal along with the mnemonic meaning of the bit position.
! The status variable IBERR will be printed in decimal along with the
! mnemonic meaning of the decimal value.  The status variable IBCNT will
! be printed in decimal.
!
! The NI-488 function IBONL is called to disable the hardware and software.
! ============================================================
SUB gpiberr

  PRINT msg$

  PRINT "ibsta = &H"; hex$(ibsta_i)

  IF b_AND(ibsta_i,-32768) <> 0 then PRINT " ERR"
  IF b_AND(ibsta_i,16384) <> 0 then PRINT " TIMO"
  IF b_AND(ibsta_i,8192) <> 0 then PRINT " END"
  IF b_AND(ibsta_i,4096) <> 0 then PRINT " SRQI"
  IF b_AND(ibsta_i,2048) <> 0 then PRINT " RQS"
  IF b_AND(ibsta_i,256) <> 0 then PRINT " CMPL"
  IF b_AND(ibsta_i,128) <> 0 then PRINT " LOK"
  IF b_AND(ibsta_i,64) <> 0 then PRINT " REM"
  IF b_AND(ibsta_i,32) <> 0 then PRINT " CIC"
  IF b_AND(ibsta_i,16) <> 0 then PRINT " ATN"
  IF b_AND(ibsta_i,8) <> 0 then PRINT " TACS"
  IF b_AND(ibsta_i,4) <> 0 then PRINT " LACS"
  IF b_AND(ibsta_i,2) <> 0 then PRINT " DTAS"
  IF b_AND(ibsta_i,1) <> 0 then PRINT " DCAS"
  PRINT

  PRINT "iberr = ", iberr
  IF iberr = 0 then PRINT " EDVR <DOS Error>"
  IF iberr = 1 then PRINT " ECIC <Not CIC>"
  IF iberr = 2 then PRINT " ENOL <No Listener>"
```

```
   IF iberr = 15 then PRINT " ESTB <Status byte lost>"
   IF iberr = 16 then PRINT " ESRQ <SRQ stuck on>"
   IF iberr = 20 then PRINT " ETAB <Table Overflow>"
   PRINT

   PRINT "ibcnt = ", ibcnt

! Call the IBONL function to disable the hardware and software.

   LET v = 0
   CALL ibonl(brd0, v)

END SUB

! ============================================================
!                    Subroutine DVMERR
! This subroutine will notify you that the Fluke 45 returned an invalid
! serial poll response byte.  The error message will be printed along with
! the serial poll response byte.
!
! The NI-488 function IBONL is called to disable the hardware and software.
! ============================================================
SUB dvmerr

   PRINT msg$

   PRINT "Status Byte = ", ord(rd$)

! Call the IBONL function to disable the hardware and software.

   LET v = 0
   CALL ibonl(brd0, v)

END SUB

END
```

```
! NKR BASIC Example Program - Device Functions

  DECLARE integer  ibsta, iberr, ibcnt        ! GPIB status variables
  DECLARE integer  dvm                        ! Device number
  DECLARE integer  mask                       ! Wait mask
  DECLARE integer  spr                        ! Serial poll response
  DECLARE integer  v                          ! GPIB function parameter
  DECLARE integer  m                          ! FOR loop index

! GPIBERR is an error subroutine that is called when a NI-488 function fails.
! DVMERR is an error subroutine that is called when the Fluke 45 does not
! have valid data to send.

  DECLARE sub gpiberr
  DECLARE sub dvmerr

  CALL cls_1
  PRINT "Read ten measurements from the Fluke 45..."
  PRINT

! Assign a unique identifier to the Fluke 45 and store in the variable
! DVM.  The name "DVM" is the name you configured for the Fluke 45 using
! IBCONF.EXE.  If DVM is less than zero, call GPIBERR with an error message.

  LET devname$ = "DVM"
  CALL ibfind (devname$,dvm)
  IF dvm < 0 then
   LET msg$ = "Ibfind Error"
   CALL gpiberr
   STOP
  END IF

! Clear the internal or device functions of the Fluke 45.  If the error bit
! ERR is set in IBSTA, call GPIBERR with an error message.

  CALL ibclr (dvm)
  LET ibsta_i = ibsta
  IF b_AND(ibsta_i,-32768) <> 0 then
   LET msg$ = "Ibclr Error"
   CALL gpiberr
   STOP
  END IF

! Reset the Fluke 45 by issuing the reset (*RST) command.  Instruct the
! Fluke 45 to measure the volts alternating current (VAC) using auto-ranging
! (AUTO), to wait for a trigger from the GPIB interface board (TRIGGER 2),
! and to assert the IEEE-488 Service Request line, SRQ, when the measurement
! has been completed and the Fluke 45 is ready to send the result (*SRE 16).
! If the error bit ERR is set in IBSTA, call GPIBERR with an error message.
```

```
! Initialize the accumulator of the ten measurements to zero.

  LET sum = 0.0

! Establish FOR loop to read the ten measuements.  The variable m will
! serve as the counter of the FOR loop.

  FOR m = 1 to 10

    ! Trigger the Fluke 45.  If the error bit ERR is set in IBSTA,
    ! call GPIBERR with an error message.

      CALL ibtrg (dvm)
      LET ibsta_i = ibsta
      IF b_AND(ibsta_i,-32768) <> 0 then
       LET msg$ = "Ibtrg Error"
       CALL gpiberr
       STOP
      END IF

    ! Request the triggered measurement by sending the instruction
    ! "VAL1?".  If the error bit ERR is set in IBSTA, call GPIBERR
    ! with an error message.

      LET wrt$ = "VAL1?"
      call ibwrt(dvm, wrt$)
      LET ibsta_i = ibsta
      IF b_AND(ibsta_i,-32768) <> 0 then
       LET msg$ = "Ibwrt Error"
       CALL gpiberr
       STOP
      END IF

    ! Wait for the Fluke 45 to request service (RQS) or wait for the
    ! Fluke 45 to timeout(TIMO).  The default timeout period is 10 seconds.
    ! RQS is detected by bit position 11 (hex 800).  TIMO is detected
    ! by bit position 14 (hex 4000).  These status bits are listed under
    ! the NI-488 function IBWAIT in the Software Reference Manual.  If the
    ! error bit ERR or the timeout bit TIMO is set in IBSTA, call GPIBERR
    ! with an error message.

      LET mask = 18432                    ! RQS or TIMO
       call ibwait (dvm,mask)
      LET ibsta_i = ibsta
      IF b_AND(ibsta_i,-32768) <> 0 then
       LET msg$ = "Ibwait Error"
       CALL gpiberr
       STOP
      END IF

    ! Read the Fluke 45 serial poll status byte.  If the error bit
    ! ERR is set in IBSTA, call GPIBERR with an error message.

      call ibrsp (dvm,spr)
```

```
      ! If the returned status byte is hex 50, the Fluke 45 has valid data to
      ! send; otherwise, it has a fault condition to report. If the status
      ! byte is not hex 50, call DVMERR with an error message.

        IF spr <> 80 then
         LET msg$ = "Fluke 45 Error"
         CALL dvmerr
         STOP
        END IF

      ! Read the Fluke 45 measurement. If the error bit ERR is set in
      ! IBSTA, call GPIBERR with an error message.

        LET rd$ = repeat$(" ",10)
        CALL ibrd (dvm,rd$)
        LET ibsta_i = ibsta
        IF b_AND(ibsta_i,-32768) <> 0 then
         LET msg$ = "Ibrd Error"
         CALL gpiberr
         STOP
        END IF

      ! Print the measurement received from the Fluke 45.

        LET rd$ = Rd$(1:IBCNT-1)
        PRINT "Reading: "; Rd$
        PRINT

      ! Convert the variable RD to its numeric value and add to the
      ! accumulator.

        LET sum = sum + val(Rd$)

    NEXT m     ! Continue FOR loop until 10 measurements are read.


! Print the average of the ten readings.

    PRINT "The average of the 10 readings is ", sum/10

! Call the ibonl function to disable the hardware and software.

    LET v = 0
    CALL ibonl(dvm, v)


! ============================================================
!                    Subroutine GPIBERR
! This subroutine will notify you that a NI-488.2 function failed by
```

```
SUB gpiberr
  PRINT msg$

  PRINT "ibsta = &H"; hex$(ibsta_i)

  IF b_AND(ibsta_i, -32768) <> 0 then PRINT " ERR"
  IF b_AND(ibsta_i, 16384) <> 0 then PRINT " TIMO"
  IF b_AND(ibsta_i, 8192) <> 0 then PRINT " END"
  IF b_AND(ibsta_i, 4096) <> 0 then PRINT " SRQI"
  IF b_AND(ibsta_i, 2048) <> 0 then PRINT " RQS"
  IF b_AND(ibsta_i, 256) <> 0 then PRINT " CMPL"
  IF b_AND(ibsta_i, 128) <> 0 then PRINT " LOK"
  IF b_AND(ibsta_i, 64) <> 0 then PRINT " REM"
  IF b_AND(ibsta_i, 32) <> 0 then PRINT " CIC"
  IF b_AND(ibsta_i, 16) <> 0 then PRINT " ATN"
  IF b_AND(ibsta_i, 8) <> 0 then PRINT " TACS"
  IF b_AND(ibsta_i, 4) <> 0 then PRINT " LACS"
  IF b_AND(ibsta_i, 2) <> 0 then PRINT " DTAS"
  IF b_AND(ibsta_i, 1) <> 0 then PRINT " DCAS"
  PRINT

  PRINT "iberr = ", iberr
  IF iberr = 0 then PRINT " EDVR <DOS Error>"
  IF iberr = 1 then PRINT " ECIC <Not CIC>"
  IF iberr = 2 then PRINT " ENOL <No Listener>"
  IF iberr = 3 then PRINT " EADR <Address error>"
  IF iberr = 4 then PRINT " EARG <Invalid argument>"
  IF iberr = 5 then PRINT " ESAC <Not Sys Ctrlr>"
  IF iberr = 6 then PRINT " EABO <Op. aborted>"
  IF iberr = 7 then PRINT " ENEB <No GPIB board>"
  IF iberr = 10 then PRINT " EOIP <Async I/O in prg>"
  IF iberr = 11 then PRINT " ECAP <No capability>"
  IF iberr = 12 then PRINT " EFSO <File sys. error>"
  IF iberr = 14 then PRINT " EBUS <Command error>"
  IF iberr = 15 then PRINT " ESTB <Status byte lost>"
  IF iberr = 16 then PRINT " ESRQ <SRQ stuck on>"
  IF iberr = 20 then PRINT " ETAB <Table Overflow>"
  PRINT

  PRINT "ibcnt = ", ibcnt

!  Call the IBONL function to disable the hardware and software.

  LET v = 0
  CALL ibonl(dvm, v)

END SUB

! ============================================================
```

```
   PRINT msg$
   LET spr_i = spr
   PRINT "Returned Byte = &H"; hex$(spr_i)

!  Call the IBONL function to disable the hardware.

   LET v = 0
   CALL ibonl (dvm, v)

END SUB

END
```

```
!  NKR BASIC Example Program - NI-488.2 Subroutines

OPTION BASE 0

   DECLARE integer ibsta, iberr, ibcnt  ! GPIB status var
   DECLARE integer instruments(32)      ! Array of primar
   DECLARE integer boardindex           ! Board index
   DECLARE integer result(32)           ! Array of listen
   DECLARE integer num_listeners        ! Number of liste
   DECLARE integer limit                ! Maximum number
   DECLARE integer mask                 ! Wait mask
   DECLARE integer k                    ! FOR loop index
   DECLARE integer  v                   ! GPIB function p
   DECLARE integer SRQasserted          ! Set to indicate
   DECLARE integer fluke                ! Primary address
   DECLARE integer statusByte           ! Serial poll res
   DECLARE integer NOADDR               ! Terminate addre
   DECLARE integer NLend                ! Send NL with EO
   DECLARE integer STOPend              ! Stop the read o

!  Constants used in this application program.

   LET NOADDR = -1
   LET NLend = 1
   LET STOPend = 256
   LET boardindex = 0

   CALL cls_1

!  Our board needs to be the Controller-In-Charge in orde
!  listeners on the GPIB.  To accomplish this, the functi
!  called.  If the error bit ERR is set in IBSTA, call GP
!  an error message.

   CALL SendIFC(boardindex)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
      LET msg$ = "SendIFC Error"
      CALL gpiberr
      STOP
   END IF

!  Create an array containing all valid GPIB primary addr
!  array (INSTRUMENTS) will be given to the function Find
!  listeners.  The constant NOADDR, defined in DECL.H, si
!  of the array.

   FOR k = 0 to 30
      LET instruments(k) = k
```

```
   PRINT "Finding all listeners on the bus..."
   LET limit = 31

   CALL FindLstn (boardindex, instruments(0), result(0),
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
      LET msg$ = "FindLstn Error"
      CALL gpiberr
      STOP
   END IF

!  Assign the value of IBCNT to the variable NUM_LISTENER
!  interface board is detected as a listener on the bus;
!  not included in the final count of the number of liste
!  the number of listeners found.

   LET num_listeners = ibcnt - 1
   PRINT "No. of instruments found = ", num_listeners

!  Send the *IDN? command to each device that was found.
!  board is at address 0 by default.  The board does not
!  skip it.
!
!  Establish a FOR loop to determine if the Fluke 45 is a
!  GPIB.  The variable LOOP will serve as a counter for t
!  as the index to the array RESULT.

   FOR k = 1 to num_listeners

        !  Send the identification query to each listen a
        !  array RESULT.  The constant NLend, defined in
        !  the function Send to append a linefeed charact
        !  to the end of the message.  If the error bit E
        !  call GPIBERR with an error message.

           LET cmd$ = "*IDN?"
           CALL Send(boardindex, result(k), cmd$, NLend)
           LET ibsta_i = ibsta
           IF b_AND(ibsta_i,-32768) <> 0 then
             LET msg$ = "Send Error"
             CALL gpiberr
             STOP
           END IF

        !  Read the name identification response returned
        !  Store the response in the array BUFFER.  The c
        !  defined in DECL.H, instructs the function Rece
        !  read when END is detected.  If the error bit E
        !  call GPIBERR with an error message.
```

```
        !  The low byte of the listen address is the prim
        !  Assign the variable PAD the primary address of
        !  The macro GetPAD, defined in DECL.H, returns t
        !  of the listen address.

           LET num_i = result(k)
           LET pad_i = b_AND(num_i,255)

        !  Print the measurement received from the Fluke

           LET rd$ = reading$(1:IBCNT-1)
           PRINT "The instrument at address ";pad_i; " is

        !  Determine if the name identification is the Fl
        !  the FLuke 45, assign PAD to FLUKE,  print mess
        !  FLuke 45 has been found, call the function FOU
        !  FOR loop.

           IF left$(Reading$, 9)="FLUKE, 45" then GOSUB 2
   NEXT k

   PRINT "Did not find the Fluke!"
   GOSUB 4000


!  Device Found.

2000
   PRINT "**** We found the Fluke 45 ****"
   LET fluke = result(k)


!  Reset the Fluke 45 using the functions DevClear and Se
!
!  DevClear will send the GPIB Selected Device Clear (SDC
!  to the Fluke 45.  If the error bit ERR is set in IBSTA
!  an error message.

   CALL DevClear (boardindex, fluke)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "DevClear Error"
     CALL gpiberr
     STOP
   END IF

!  Use the function Send to send the IEEE-488.2 reset com
!  to the Fluke 45.  The constant NLend, defined in DECL.
```

```
     CALL gpiberr
     STOP
   END IF

!  Use the function Send to send device configuration com
!  Fluke 45.  Instruct the Fluke 45 to measure volts alte
!  (VAC) using auto-ranging (AUTO), to wait for a trigger
!  interface board (TRIGGER 2), and to assert the IEEE-48
!  line, SRQ, when the measurement has been completed and
!  ready to send the result (*SRE 16).  If the error bit
!  IBSTA, call GPIBERR with an error message.

   LET cmd$ = "VAC; AUTO; TRIGGER 2; *SRE 16"
   CALL Send(boardindex, fluke, cmd$, NLend)
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "Send setup Error"
     CALL gpiberr
     STOP
   END IF

!  Initialized the accumulator of the ten measurements to

   LET sum = 0

!  Establish FOR loop to read the ten measuements.  The v
!  serve as the counter of the FOR loop.

   FOR m = 1 to 10

        !  Trigger the Fluke 45 by sending the trigger co
        !  request a measurement by sending the command "
        !  error bit ERR is set in IBSTA, call GPIBERR wi

           LET cmd$ = "*TRG; VAL1?"
           CALL Send(boardindex, fluke, cmd$, NLend)
           LET ibsta_i = ibsta
           IF b_AND(ibsta_i,-32768) <> 0 then
             LET msg$ = "Send trigger Error"
             CALL gpiberr
             STOP
           END IF

        !  Wait for the Fluke 45 to assert SRQ, meaning i
        !  a measurement.  If SRQ is not assserted within
        !  call GPIBERR with an error message.  The timeo
        !  is 10 seconds.

           CALL WaitSRQ(boardindex, SRQasserted)
```

```
!  Read the serial poll status byte of the FLuke
!  bit ERR is set in IBSTA, call GPIBERR with an

   CALL ReadStatusByte(boardindex, fluke, statusB
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "ReadStatusByte Error"
     CALL gpiberr
     STOP
   END IF

!  Check if the Message Available Bit (bit 4) of
!  byte is set.  If this bit is not set, print th
!  call GPIBERR with an error message.

   LET mask = 16
   LET status_i = statusByte
   IF b_AND(status_i,mask) <> 16 then
     LET msg$ = "Improper Status Byte"
     CALL gpiberr
     PRINT "Status Byte = "; statusByte
     STOP
   END IF

!  Read the Fluke 45 measurement.  Store the meas
!  variable BUFFER.  The constant STOPend, define
!  instructs the function Receive to terminate th
!  is detected.  If the error bit ERR is set in I
!  GPIBERR with an error message.

   CALL Receive (boardindex, fluke, Reading$, STO
   LET ibsta_i = ibsta
   IF b_AND(ibsta_i,-32768) <> 0 then
     LET msg$ = "Receive Error"
     CALL gpiberr
     STOP
   END IF

!  Use the null character to mark the end of the
!  in the array BUFFER.  Print the measurement re
!  Fluke 45.

   LET rd$ = Reading$(1:IBCNT-1)
   PRINT "Reading : ";rd$
   PRINT

!  Convert the variable BUFFER to its numeric val
!  accumulator.
```

```
! ========================================================
!                       Subroutine GPIBERR
!  This function will notify you that a NI-488.2 function
!  printing an error message.  The status variable IBSTA
!  printed in hexadecimal along with the mnemonic meaning
!  The status variable IBERR will be printed in decimal a
!  mnemonic meaning of the decimal value.  The status var
!  be printed in decimal.
!
!  The NI-488 function IBONL is called to disable the har
! ========================================================
SUB gpiberr
   PRINT msg$

   PRINT "ibsta = &H"; hex$(ibsta_i)

   IF b_AND(ibsta_i, -32768) <> 0 then PRINT " ERR"
   IF b_AND(ibsta_i, 16384) <> 0 then PRINT " TIMO"
   IF b_AND(ibsta_i, 8192) <> 0 then PRINT " END"
   IF b_AND(ibsta_i, 4096) <> 0 then PRINT " SRQI"
   IF b_AND(ibsta_i, 2048) <> 0 then PRINT " RQS"
   IF b_AND(ibsta_i, 256) <> 0 then PRINT " CMPL"
   IF b_AND(ibsta_i, 128) <> 0 then PRINT " LOK"
   IF b_AND(ibsta_i, 64) <> 0 then PRINT " REM"
   IF b_AND(ibsta_i, 32) <> 0 then PRINT " CIC"
   IF b_AND(ibsta_i, 16) <> 0 then PRINT " ATN"
   IF b_AND(ibsta_i, 8) <> 0 then PRINT " TACS"
   IF b_AND(ibsta_i, 4) <> 0 then PRINT " LACS"
   IF b_AND(ibsta_i, 2) <> 0 then PRINT " DTAS"
   IF b_AND(ibsta_i, 1) <> 0 then PRINT " DCAS"
   PRINT

   PRINT "iberr = ", iberr
   IF iberr = 0 then PRINT " EDVR <DOS Error>"
   IF iberr = 1 then PRINT " ECIC <Not CIC>"
   IF iberr = 2 then PRINT " ENOL <No Listener>"
   IF iberr = 3 then PRINT " EADR <Address error>"
   IF iberr = 4 then PRINT " EARG <Invalid argument>"
   IF iberr = 5 then PRINT " ESAC <Not Sys Ctrlr>"
   IF iberr = 6 then PRINT " EABO <Op. aborted>"
   IF iberr = 7 then PRINT " ENEB <No GPIB board>"
   IF iberr = 10 then PRINT " EOIP <Async I/O in prg>"
   IF iberr = 11 then PRINT " ECAP <No capability>"
   IF iberr = 12 then PRINT " EFSO <File sys. error>"
   IF iberr = 14 then PRINT " EBUS <Command error>"
   IF iberr = 15 then PRINT " ESTB <Status byte lost>"
   IF iberr = 16 then PRINT " ESRQ <SRQ stuck on>"
   IF iberr = 20 then PRINT " ETAB <Table Overflow>"
   PRINT

   PRINT "ibcnt = ", ibcnt
```