NI-488® and NI-488.2™
Subroutines for Pascal

November 1993 Edition

Part Number 320375-01

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

Branch Offices:
Australia (03) 879 9422, Austria (0662) 435986, Belgium 02/757.00.20,
Canada (Ontario) (519) 622-9310, Canada (Québec) (514) 694-8521,
Denmark 45 76 26 00, Finland (90) 527 2321, France (1) 48 14 24 24,
Germany 089/741 31 30, Italy 02/48301892, Japan (03) 3788-1921,
Netherlands 03480-33466, Norway 32-848400, Spain (91) 640 0085,
Sweden 08-730 49 70, Switzerland 056/20 51 51, U.K. 0635 523545

Limited Warranty

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

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

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

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

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

Copyright

Trademarks

Warning Regarding Medical and Clinical Use of National Instruments Products

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

Preface

This manual contains information for programming the NI-488.2 routines and the NI-488 functions in Pascal. The term *Pascal* as used in this manual, includes IBM Pascal, Microsoft Pascal, Microsoft QuickPascal, Turbo Pascal and Turbo Pascal for Windows.

This manual assumes that the software is installed and that you are familiar with the software operation. Programming knowledge in a Pascal language and familiarity with the compiler are also assumed.

Organization of This Manual

This manual is organized as follows:

- Chapter 1, *General Information*, lists the terms and mnemonics used in this manual, lists the distribution files relevant to programming in Pascal languages, and explains programming preparations. This chapter also discusses several special functions and parameters and summarizes how to use the NI-488.2 routine and NI-488 function calls that are 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 examples. The descriptions are listed alphabetically for easy reference.

- Chapter 3, *NI-488 Function Descriptions*, contains a detailed description of each NI-488 function with examples. 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*, explains how to use, install, and configure the Applications Monitor, a resident program that is useful in debugging sequences of NI-488 and NI-488.2 calls from within your MS-DOS application.

- Appendix C, *Customer Communication*, contains forms for you to complete to facilitate communication with National Instruments concerning our products.

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 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.

The following terms are used in this manual:

Term	Reference
IBM/MS Pascal	IBM and Microsoft Pascal
QuickPascal	Microsoft QuickPascal
Turbo Pascal	Borland Turbo Pascal
Turbo Pascal for Windows	Borland Turbo Pascal for Windows
Pascal	All the Pascal languages supported
MS-DOS Pascal	Excludes Borland Turbo Pascal for Windows

Abbreviations

The following metric system prefixes are used with abbreviations for units of measure in this manual:

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

The following abbreviations are used in this manual:

Hz	hertz
hex	hexadecimal
M	megabytes
sec	second

Acronyms

The following acronyms are used in this manual:

AC	alternating current
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Exchange
DIO	digital input/output
DMA	direct memory access
DVM	digital voltmeter
GPIB	General Purpose Interface (IEEE-488) bus
IEEE	Institute of Electrical and Electronic Engineers
I/O	input/output
PC	personal computer
RAM	random-access memory
T/L	Talker/Listener
VAC	volts alternating current

Related Documents

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

- *NI-488.2 Software Reference Manual for MS-DOS* (National Instruments Corporation 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, Protocols, and Common Commands*

Customer Communication

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

Contents

Figure

Tables

Chapter 1
General Information

This chapter lists the terms and mnemonics used in this manual, lists the distribution files relevant to programming in Pascal languages, and explains programming preparations. This chapter also discusses several special functions and parameters and summarizes how to use the NI-488.2 routine and NI-488 function calls that are explained at length in Chapter 2 and Chapter 3.

Terms and Mnemonics Used in This Manual

Table 1-1 lists the terms used in this manual.

Table 1-1. Terms Used in This Manual

Term	Reference
IBM/MS Pascal	IBM and Microsoft Pascal
QuickPascal	Microsoft QuickPascal
Turbo Pascal	Borland Turbo Pascal
Turbo Pascal for Windows	Borland Turbo Pascal for Windows
Pascal	All the Pascal languages supported
MS-DOS Pascal	Excludes Borland Turbo Pascal for Windows

Table 1-2 lists the mnemonics used in this manual.

Table 1-2. Mnemonics Used in This Manual

Mnemonic	Full Name
ATN	Attention
CACS	Controller Active State
CIC	Controller-In-Charge
CIDS	Controller Idle State
CMPL	Complete
DAV	Data Valid
DCAS	Device Clear Active State
DTAS	Device Trigger Active State
EOI	end or identify
EOS	end of string
ERR	Error Bit
IDY	Identify
IFC	Interface Clear
LACS	Listener Active State
LOK	Lockout Bit
LPE	Local Poll Enable
MAV	Message Available
NDAC	Not Data Accepted
NRFD	Not Ready For Data
REM	Remote Bit
REN	Remote Enable
RST	Reset
RQS	Request Service
RTL	Return To Local
SPOLL	Serial Poll
SRQ	Service Request
SRQI	Service Request Input Bit
TACS	Talker Active State
TIMO	Timeout

Using the Distribution Files

Your kit includes supplemental distribution disks for either IBM/MS Pascal and QuickPascal, Turbo Pascal, or Turbo Pascal for Windows. Each supplemental disk contains files relevant to programming in the corresponding Pascal language. Copy the distribution files that you need to your work area and store the originals in a safe place.

IBM Pascal and Microsoft Pascal Files

The *NI-488.2 Supplemental Disk for MS-DOS Driver IBM/MS Pascal and MS QuickPascal Language Interface* contains five files relevant to programming in IBM/MS Pascal:

- `DECL.PAS` is a file containing declarations.
- `PIB.OBJ` is the Pascal language interface that gives your application program access to the software.
- `DPSAMP.PAS` is a sample program using device calls.
- `BPSAMP.PAS` is a sample program using board calls.
- `PSAMP488.PAS` is a sample program using NI-488.2 calls.

Microsoft QuickPascal Files

The *NI-488.2 Supplemental Disk for MS-DOS Driver IBM/MS Pascal and MS QuickPascal Language Interface* contains five files relevant to programming in QuickPascal:

- `QPDECL.PAS` is a file containing declarations.
- `PIB.OBJ` is the Pascal language interface that gives your application program access to the software.
- `DQPSAMP.PAS` is a sample program using device calls.
- `BQPSAMP.PAS` is a sample program using board calls.
- `QSAMP488.PAS` is a sample program using NI-488.2 calls.

Turbo Pascal Files

The *NI-488.2 Supplemental Disk for MS-DOS Driver Turbo Pascal Language Interface* contains six files relevant to programming in Turbo Pascal:

- `TPDECL.TPU` is a unit containing required variable, procedure, and constant declarations and initializations. Compiled for Turbo Pascal 6.0.
- `TPDECL.PAS` is the Turbo Pascal source code for `TPDECL.TPU`.
- `TPIB.OBJ` is the Pascal language interface that gives your application program access to the software.
- `DTPSAMP.PAS` is a sample program using device calls.
- `BTPSAMP.PAS` is a sample program using board calls.
- `TSAMP488.PAS` is a sample program using NI-488.2 calls.

Turbo Pascal for Windows Files

The *NI-488.2 Supplemental Disk for MS-Windows Turbo Pascal for Windows Language Interface* contains five files relevant to programming in Turbo Pascal for Windows:

- `TPWGPIB.TPU` is a unit containing constant declaration and initialization, function and procedure prototypes, and direct entry points to the dynamic link library `gpib.dll`.
- `TPWGPIB.PAS` is the Turbo Pascal for Windows source code for `TPWGPIB.TPU`.
- `DTPWSAMP.PAS` is the sample program using device calls.
- `BTPWSAMP.PAS` is the sample program using board calls.
- `WSAMP488.PAS` is the sample program using NI-488.2 calls.

Programming Preparations

After you have copied the distribution files that you need to your work area, include the declaration files in your application program by following the instructions for your particular Pascal language.

MS-DOS Pascal Preparations

In the MS-DOS Pascal programming languages, array variables defined by the user must be of the same type as the parameters in the subroutine declarations. In order to conform to this strict type checking, array types have been defined in the header files (`DECL.PAS`, `QPDECL.PAS`, and `TPDECL.PAS`) for typecasting file name parameters, board or device name parameters, and command buffer parameters. You must declare these types of variables as follows:

```
file name buffers    typed  flbuf(array[1..50] of char)
board/device buffers        typed nbuf(array[1..7] of
char)
command buffers      typed  cbuf(array[1..255] of char)
```

Note: Board or device names must be arrays of seven characters. Use blanks to pad the name.

Example:
```
var bdname : nbuf;
bdname := 'GPIB0 ';
```

IBM/MS Pascal Preparations

Place the following IBM/MS Pascal statement at the beginning of your application program to include `DECL.PAS`:

```
{$INCLUDE: 'DECL.PAS'}
```

The file `PIB.OBJ` is the IBM/MS Pascal language interface to the NI-488.2 software for MS-DOS. Link the file `PIB.OBJ` to the GPIB application program written in Pascal to produce an executable file.

Note: The file `PIB.OBJ` must *not* be the first file named in the link when linking with the application program.

QuickPascal Preparations

Place the following QuickPascal directive at the beginning of your application program to include QPDECL.PAS:

```
{$I QPDECL.PAS}
```

The LINK directive is used in QPDECL.PAS to link PIB.OBJ with your application program. The FAR CALLS compiler option is enabled in TPDECL.PAS.

Turbo Pascal Preparations

Place the following Turbo Pascal statement at the beginning of your application program:

```
uses TPDECL;
```

If you have Turbo Pascal 5.x, compile TPDECL.PAS to create a new TPDECL.TPU for your compiler. The file TPDECL.TPU contains essential Pascal declarations and initializations. The LINK directive is used in TPDECL.PAS to link PIB.OBJ with your application program. Do not *redeclare* global variables in your application program if they are declared in TPDECL.PAS.

The formal parameters that correspond to read or write buffers are untyped parameters. You can define new data types for these buffers or use type cbuf (defined in TPDECL.PAS as `array [1..255] of char;`).

Use the command tpc to create TPDECL.TPU or to create the executable form of your application program, as shown in the following examples:

```
Create TPDECL.TPU : tpc tpdecl.pas
```

or

```
Create.exe fle : tpc yourprogramname.pas
```

Turbo Pascal for Windows Preparations

Place the following Turbo Pascal for Windows statement at the beginning of your application program:

```
uses TPWGPIB;
```

The formal parameters for board or device names and filenames are of type `PChar`. Therefore, the corresponding actual parameters must be of type `PChar`.

Example:
```
var ud:integer
bdname:PChar;
bdname:='GPIB0';
ud:=ibfind(bdname);

(* or  ud := ibfind('GPIB0');)
```

Because formal parameters for command, read, and write buffers are untyped, Turbo Pascal variables of a user-specified array type or of type `cbuf` are compatible with the Turbo Pascal for Windows language interface. Turbo Pascal application programs that use parameter types `nbuf` and `flbuf` must be changed to use the Turbo Pascal for Windows predefined array type `PChar`.

Note: The Turbo Pascal for Windows language interface cannot be used for protected-mode application programs.

Using ibsrq for "ON SRQ" Capability

Pascal application programs can be interrupted by the GPIB SRQ signal. You can cause the program to go to a user-specified service routine when the interrupt occurs. A special board function, `ibsrq`, receives the address of the user-specified service routine and makes the NI-488.2 for MS-DOS driver check for the occurrence of an SRQ after each GPIB function has completed. When SRQ is asserted, `ibsrq` sends the program to the user-specified routine. Refer to the *`IBSRQ`* function in Chapter 3 for a complete description and programming examples. `ibsrq` is not supported in Turbo Pascal for Windows.

Using ibsta to Test 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 error
```

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

You can also test for particular bits in the status word. The following is an example of testing for the CMPL bit:

```
if ((ibsta AND CMPL) <> 0) then...
```

Note: Explicit code that tests the status word is not necessary in MS-DOS Pascal if you are using the applications monitor. For information on the applications monitor refer to Appendix B.

Using ibcnt and ibcntl as Count Variables

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) and `ibcntl` is a long integer value (32 bits wide). `ibcntl` is not available for IBM Pascal application programs.

Using I/O Variable Parameters

The variables you use in input/output calls must be consistent with the parameters defined for your particular Pascal language. The parameters are defined in the declaration header files for each Pascal language.

MS-DOS Pascal I/O Variable Parameters

The most commonly used I/O calls are `ibrd` and `ibwrt`. In IBM/MS Pascal and QuickPascal these functions read and write from a character array of type `cbuf`. The following declaration is in the header files `DECL.PAS` and `QPDECL.PAS`:

```
cbuf = array [1..255] of char;
```

In addition, integer I/O calls (`ibrdi` and `ibwrti` for example) are for users 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. The integer array type for integer I/O calls, `ibuf`, is defined in the header files `DECL.PAS` and `QPDECL.PAS` as follows:

```
ibuf = array [1..512] of integer;
```

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. The asynchronous functions `ibrdia` and `ibwrtia` perform asynchronous integer reads and writes.

Note: `ibrdi` and `ibwrti` are not necessary in Turbo Pascal. `ibrd` and `ibwrt` receive data of a user-defined integer or character array of the type `cbuf` (defined in `TPDECL.PAS` as `array[1..255] of char`).

Turbo Pascal for Windows I/O Variable Parameters

The I/O calls `ibrd` and `ibwrt` receive data of either a user-defined integer or character array, type `cbuf` (defined in `TPWGPIB.PAS` as `array[1..255] of char`), or type `PChar`. Literal strings cannot be used as actual parameters in the subroutine call because the corresponding formal parameters are untyped. If a variable is declared as type `PChar`, it must be referenced in the subroutine call statement by using the pointer symbol (`^`) after the variable name.

Example:

```
var  wrt: PChar;
wrt := 'VAL1?';
ibwrt(dvm,wrt^,5);
```

User-defined arrays and arrays of type `cbuf` are referenced by name only. Using the pointer symbol (`^`) after these variables is an error.

Pascal NI-488.2 I/O Variable Parameters

For the NI-488.2 subroutines requiring arrays of GPIB addresses, the following array type is defined in the header files `DECL.PAS`, `QPDECL.PAS`, `TPDECL.PAS`, and `TPWGPIB.PAS`:

```
addrlist : array[0..255] of integer;
```

All the parameters of the NI-488.2 subroutines indicated in Chapter 2 of this manual as arrays of GPIB addresses must be declared as type `addrlist`. You must use the value `NOADDR` (defined in the header files as `-1`) to terminate an array of addresses that is being sent over the GPIB.

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

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 first argument of all calls and functions except `ibdev`, `ibfind`, `ibsrq`, and `ibtrap` is the integer variable `ud`. This serves as a general unit descriptor to show the format of the calls. In practice, `ud` refers to the board or device to which the command is directed. Refer to the *IBFIND* and *IBDEV* function descriptions in this chapter and to Chapter 3 to determine the type of unit descriptor to use.

The routines and functions are listed alphabetically by name in Chapters 2 and 3, respectively. Tables 1-3 and 1-4 list the Pascal NI-488.2 routines and NI-488 functions, respectively, along with a brief descriptions of each routine and function. The format is identical for IBM/MS Pascal, QuickPascal, Turbo Pascal, and Turbo Pascal for Windows.

Table 1-3. Pascal NI-488.2 Routines

Call Syntax	Description
`AllSpoll (board,addresslist, resultlist)`	Serial poll all devices
`DevClear (board,address)`	Clear a single device
`DevClearList (board,addresslist)`	Clear multiple devices
`EnableLocal (board,addresslist)`	Enable operations from the front of a device
`EnableRemote (board,addresslist)`	Enable remote GPIB programming of devices
`FindLstn (board,addresslist, resultlist,limit)`	Find all Listeners
`FindRQS (board,addresslist,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, sense)`	Configure a device for parallel polls
`PPollUnconfig (board,addresslist)`	Unconfigure devices for parallel polls
`RcvRespMsg (board,data,count, 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,count, 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)`	Initialize a GPIB system on three levels

(continues)

Table 1-3. Pascal NI-488.2 Routines (continued)

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

Table 1-4 gives a brief description of each NI-488 function. The first argument of all function calls except `ibfind`, `ibdev`, `ibsrq`, and `ibtrap` is the integer variable `ud`, which serves as a unit descriptor. Refer to the *IBFIND* and *IBDEV* function descriptions in Chapter 3 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 1-4. Pascal 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,cnt)`	Send commands from string
`ibcmda (ud,cmd,cnt)`	Send commands asynchronous from string
`ibconfig (ud,option,value)`	Configure the software
`ibdev (bdindex,pad,sad,tmo, eot,eos)`	Open an unused device when device name is unknown
`ibdma (ud,v)`	Enable/disable DMA
`ibeos (ud,v)`	Change/disable EOS mode
`ibeot (ud,v)`	Enable/disable END message (write)
`ibevent (ud,event)`	Return the next event
`ibfind(udname)`	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 for GPIB lines
`ibln (ud,pad,sad,listen)`	Check for presence of a device on 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,cnt)`	Read data to string
`ibrda (ud,rd,cnt)`	Read data asynchronously to string
`ibrdf (ud,flname)`	Read data to file

(continues)

Table 1-4. Pascal NI-488 Functions (continued)

Call Syntax	Description
`ibrdi (ud,iarr,cnt)`	Read data to integer array
`ibrdia (ud,iarr,cnt)`	Read data asynchronously 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 status byte
`ibsad (ud,v)`	Change Secondary Address
`ibsic (ud)`	Send Interface Clear for 100 µsec
`ibsre (ud,v)`	Set/clear Remote Enable line
`ibsrq (@ func)`	Register an SRQ "interrupt routine"
`ibstop (ud)`	Abort asynchronous operation
`ibtmo (ud,v)`	Change/disable time limit
`ibtrap (mask,mode)`	Alter applications monitor trap and display modes
`ibtrg (ud)`	Trigger selected device
`ibwait (ud,mask)`	Wait for selected event
`ibwrt (ud,wrt,cnt)`	Write data from string
`ibwrta (ud,wrt,cnt)`	Write data asynchronously from string
`ibwrtf (ud,flname)`	Write data from file
`ibwrti (ud,iarr,cnt)`	Write data from integer array
`ibwrtia (ud,iarr,cnt)`	Write data asynchronously from integer array

Using Functions that Reconfigure Board or 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 1-5.

Table 1-5. Functions That Alter Default Characteristics

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

Chapter 2
NI-488.2 Routine Descriptions

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

AllSpoll AllSpoll

Purpose: Serial Poll all devices.

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

`board` represents a board number. The parameters `addresslist` and `resultlist` are arrays of type `addrlist`, as defined in the declaration files. `addresslist` must be terminated by the value `NOADDR`. 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, 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`.

```
var addresslist : addrlist;
    resultlist  : addrlist;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := NOADDR;
AllSpoll (0,addresslist,resultlist);
```

DevClear DevClear

Purpose: Clear a single device.

Format: `DevClear (board,address)`

`board` represents 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`, 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`.

```
var address : integer;
address := 9 + 256 * 97;
DevClear (0,address);
```

DevClearList DevClearList

Purpose: Clear multiple devices.

Format: `DevClearList (board,addresslist)`

`board` represents a board number. The GPIB devices whose addresses are contained in the address array are cleared. The parameter `addresslist` is an array of type `addrlist`, as defined in the declaration files, and must be terminated by the value `NOADDR`.

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 the value `NOADDR`, the universal Device Clear message is sent to all devices on the GPIB.

Example:

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

```
var addresslist : addrlist;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := NOADDR;
DevClearList (0,addresslist);
```

EnableLocal EnableLocal

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

Format: `EnableLocal (board,addresslist)`

`board` represents a board number. The GPIB devices whose addresses are contained in the `addresslist` array are placed in local mode by addressing them as Listeners and sending the GPIB Go To Local command. The parameter `addresslist` is an array of type `addrlist`, as defined in the declaration files, and must be terminated by the value `NOADDR`.

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.

```
var addresslist : addrlist;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := NOADDR;
EnableLocal (0,addresslist);
```

EnableRemote EnableRemote

Purpose: Enable remote GPIB programming of devices.

Format: `EnableRemote (board,addresslist)`

`board` represents 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 of type `addrlist`, as defined in the declaration files, and must be terminated by the value `NOADDR`.

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

Example:

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

```
var addresslist : addrlist;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := NOADDR;
EnableRemote (0,addresslist);
```

FindLstn FindLstn

Purpose: Find all Listeners.

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

`board` represents a board number. `addresslist` contains a list of primary GPIB addresses, terminated by the value `NOADDR`. 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. `addresslist` and `resultlist` are of type `addrlist`, as defined in the declaration files.

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 can usually be interrogated by identification messages to determine what devices they are.

FindLstn (continued) FindLstn

Example:

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

```
var addresslist : addrlist;
    resultlist  : addrlist;
    limit       : integer;

(* Because there are three primary GPIB           *)
(* addresses, 93 separate GPIB devices could      *)
(* be detected at all the secondary addresses.    *)
(* This example assumes that there are, at        *)
(* most, five devices connected to the GPIB.      *)

addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := 10;
addresslist[3] := NOADDR;
limit := 5;
FindLstn (0,addresslist,resultlist,limit);
```

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

```
resultlist [0]   9
resultlist [1]   10 + 96*256
resultlist [2]   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: `FindRQS (board,addresslist,result)`

`board` represents a board number. `addresslist` contains a list of primary GPIB addresses. 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`. `addresslist` is of type `addrlist`, as defined in the declaration files, and must be terminated by the value `NOADDR`.

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` is requesting service.

```
var addresslist : addrlist;
         result : integer;
result := 0;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := 10;
addresslist[3] := NOADDR;
FindRQS (0,addresslist,result);
```

Following this call, `result` might contain the value hex 40 (the serial poll response), and `ibcnt` might contain the value `2`, 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: `PassControl (board,address)`

`board` represents 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`.

```
PassControl (0,9);
```

PPoll PPoll

Purpose: Perform a parallel poll.

Format: `PPoll (board,result)`

`board` represents 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`.

```
var result : integer;
PPoll (0,result);
```

PPollConfig PPollConfig

Purpose: Configure a device for parallel polls.

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

`board` represents 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).

```
PPollConfig (0,8,5,0);
```

PPollUnconfig PPollUnconfig

Purpose: Unconfigure devices for parallel polls.

Format: `PPollUnconfig (board,addresslist)`

`board` represents a board number. The GPIB devices whose addresses are contained in the `addresslist` array are unconfigured for parallel polls–that is, they no longer participate in polls. The parameter `addresslist` is an array of type `addrlist`, as defined in the declaration files, and must be terminated by the value `NOADDR`.

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`.

```
var addresslist : addrlist;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := NOADDR;
PPollUnconfig (0,addresslist);
```

RcvRespMsg RcvRespMsg

Purpose: Read instruction bytes from already addressed device.

Format: `RcvRespMsg (board,data,count,termination)`

`board` represents a board number. Up to `count` instruction bytes are read from the GPIB and placed into the pre-allocated array `data`. The `count` argument is of type `integer4` in Microsoft Pascal and is of type `longINT` in QuickPascal, Turbo Pascal, and Turbo Pascal for Windows. `count` is of type `integer` in IBM Pascal. `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 files `DECL.PAS`, `QPDECL.PAS`, and `TPDECL.PAS`), 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 devices and then receiving instruction bytes.

Example:

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

Pascal

```
var data : cbuf;
RcvRespMsg (0,data,100,10);
```

Turbo Pascal for Windows

```
var data : array[0..99] of char;
RcvRespMsg(0,data,100,10);
(* input into a null-terminated string             *)
```

ReadStatusByte ReadStatusByte

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

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

`board` represents a board number. The indicated device is serial polled, and its status byte is placed into the variable `result`. Only the lower byte of `result` is affected. The upper byte contains whatever value it did before the call was made.

Example:

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

```
var result : integer;
ReadStatusByte (0,8,result);
```

Receive Receive

Purpose: Read instruction bytes from a GPIB device.

Format: `Receive (board,address,data,count,termination)`

`board` represents a board number. The GPIB device at the indicated `address` is addressed, and up to `count` instruction bytes are read from that device and placed into a pre-allocated array `data`. The `count` value is of type `integer4` in Microsoft Pascal and of type `longINT` in QuickPascal, Turbo Pascal, and Turbo Pascal for Windows. Even though it is a long value in these languages, integer values may also be passed. `count` is of type `integer` in IBM Pascal. `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` (defined in the declaration files), 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.

Pascal

```
var data : cbuf;
Receive (0,8,data,100,STOPend);
```

Turbo Pascal for Windows

```
var data : array[0..100] of char;
Receive(0,8,data,100,STOPend);
(* input into a null-terminated string             *)
```

ReceiveSetup ReceiveSetup

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

Format: `ReceiveSetup (board,address)`

`board` represents 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. It can be 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 instruction bytes to board `0`. Then, receive messages of up to 100 bytes from the device. The message is terminated with END.

Pascal

```
var message : cbuf;
ReceiveSetup (0,8);
RcvRespMsg (0,message,100,STOPend);
```

Turbo Pascal for Windows

```
var message : array[0..100] of char;
ReceiveSetup(0,8);
RcvRespMsg(0,message,100,STOPend);
(* input into a null-terminated string             *)
```

ResetSys ResetSys

Purpose: Initialize a GPIB system on three levels.

Format: `ResetSys (board,addresslist)`

`board` represents a board number. You must terminate `addresslist` with the value `NOADDR`. `ResetSys` initializes the GPIB system 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 IEEE-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`.

```
var addresslist : addrlist;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := 10;
addresslist[3] := NOADDR;
ResetSys (0,addresslist);
```

Send Send

Purpose: Send instruction bytes to a single GPIB device.

Format: `Send (board,address,data,count,eotmode)`

`board` represents a board number. The GPIB device is addressed as a Listener, the indicated board is addressed as a Talker, and `count` instruction bytes contained in the pre-allocated array `data` are sent. The `count` value is of type `integer4` in Microsoft Pascal and of type `longINT` in QuickPascal, Turbo Pascal, and Turbo Pascal for Windows. `count` is of type `integer` in IBM Pascal. `eotmode` is a flag used to describe the method of signaling the end of the data to the Listener. Set it to one of the following constants:

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

These constants are defined in the declaration files `DECL.PAS`, `QPDECL.PAS`, `TPDECL.PAS`, and `TPWGPIB.PAS`.

Example:

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

Pascal

```
var data : cbuf;
data[1] := '*';
data[2] := 'I';
data[3] := 'D';
data[4] := 'N';
data[5] := '?';
Send (0,8,data,5,NLend);
```

Turbo Pascal for Windows

```
var data : PChar;
data := '*IDN?';
Send(0,8,data^,5,NLend);
```

SendCmds SendCmds

Purpose: Send GPIB command bytes.

Format: `SendCmds (board,commands,count)`

`board` represents a board number. The pre-allocated array `commands` contains command bytes to be sent onto the GPIB. The number of bytes to be sent from the string is indicated by the argument `count`. The `count` value is of type `integer4` in Microsoft Pascal and of type `longINT` in QuickPascal, Turbo Pascal, and Turbo Pascal for Windows. `count` is of type `integer` in IBM Pascal.

`SendCmds` is not normally required for GPIB operation. Use it 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.

IBM/MS Pascal

```
var commands : cbuf;
commands[1] := chr(16#3F);
commands[2] := chr(16#40);
commands[3] := chr(16#28);
commands[4] := chr(16#29);
commands[5] := chr(16#04);
commands[6] := chr(16#01);
SendCmds (0,commands,6);
```

QuickPascal/Turbo Pascal

```
var commands : cbuf;
commands[1] := chr($3F);
commands[2] := chr($40);
commands[3] := chr($28);
commands[4] := chr($29);
commands[5] := chr($04);
commands[6] := chr($01);
SendCmds (0,commands,6);
```

SendCmds (continued) SendCmds

Turbo Pascal for Windows

```
var commands : PChar;
commands := #3F#40#28#29#04#01;
SendCmds(0,commands^,6);
```

SendDataBytes SendDataBytes

Purpose: Send instruction bytes to already addressed devices.

Format: `SendDataBytes (board,data,count,eotmode)`

`board` represents a board number. The pre-allocated array `data` contains instruction bytes to be sent onto the GPIB. The number of bytes to be sent from the string is indicated by the argument `count`. The `count` value is of type `integer4` in Microsoft Pascal and of type `longINT` in QuickPascal, Turbo Pascal, and Turbo Pascal for Windows. Even though it is a long value in these languages, integer values may also be passed. `count` is of type `integer` in IBM Pascal. `eotmode` is a flag used to describe the method of signaling the end of the data to the Listeners. Set it to one of the following constants:

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

These constants are defined in the header files `DECL.PAS`, `QPDECL.PAS`, `TPDECL.PAS`, and `TPWGPIB.PAS`.

`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 instruction bytes.

SendDataBytes (continued) SendDataBytes

Example:

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

Pascal

```
var data : cbuf;
data[1] := '*';
data[2] := 'I';
data[3] := 'D';
data[4] := 'N';
data[5] := '?';
SendDataBytes (0,data,5,NLend);
```

Turbo Pascal for Windows

```
var data : PChar;
data := '*IDN?';
SendDataBytes(0,data^,5,NLend);
```

SendIFC SendIFC

Purpose: Clear the GPIB interface functions with IFC.

Format: `SendIFC (board)`

`board` represents a board number. When the GPIB Device IFC message is issued, the interface functions of all connected devices return 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`.

```
SendIFC (0);
```

SendList SendList

Purpose: Send instruction bytes to multiple GPIB devices.

Format:
```
SendList
(board,addresslist,data,count,eotmode)
```

`board` represents a board number. `addresslist` contains a list of primary GPIB addresses, and must be terminated by the value `NOADDR`. The GPIB devices whose addresses are contained in the address array are addressed as Listeners, the indicated board is addressed as a Talker, and `count` instruction bytes contained in the pre-allocated array `data` are sent. `addresslist` is of type `addrlist` as defined in the header files. The `count` value is of type `integer4` in Microsoft Pascal and of type `longINT` in QuickPascal, Turbo Pascal, and Turbo Pascal for Windows. `count` is of type `integer` in IBM Pascal. `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` Send NL (linefeed) with EOI after the instruction bytes.
- `DABend` Send EOI with the last data byte in the string.
- `NULLend` Do nothing to mark the end of the transfer.

These constants are defined in the header files `DECL.PAS`, `QPDECL.PAS`, `TPDECL.PAS`, and `TPWGPIB.PAS`.

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 addresses 8 and 9. The transmission should be terminated using a linefeed character with EOI.

Pascal

```
var addresslist : addrlist;
          data : cbuf;
addresslist [0] := 8;
addresslist [1] := 9;
addresslist [2] := NOADDR;
data[1] := '*';
data[2] := 'I';
data[3] := 'D';
data[4] := 'N';
data[5] := '?';
SendList (0,addresslist,data,5,NLend);
```

Turbo Pascal for Windows

```
var addresslist : addrlist;
          data : PChar;
addresslist [0] := 8;
addresslist [1] := 9;
addresslist [2] := NOADDR;
data := '*IDN?';
SendList (0,addresslist,data^,5,NLend);
```

SendLLO SendLLO

Purpose: Send the Local Lockout message to all devices.

Format: `SendLLO (board)`

`board` represents 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`.

```
SendLLO (0);
```

SendSetup SendSetup

Purpose: Prepare particular devices to receive instruction bytes.

Format: `SendSetup (board,addresslist)`

`board` represents 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 of type `addrlist`, as defined in the declaration files, and must be terminated by the value `NOADDR`.

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 instruction 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.

Pascal

```
var data         : array[1..5] of array[1..9] of
char;
   eotmode,k,i   : integer;
   addresslist   : addrlist;
      message    : cbuf;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := NOADDR;
eotmode := NULLend;
data[1] := 'Message 1';
data[2] := 'Message 2';
data[3] := 'Message 3';
data[4] := 'Message 4';
data[5] := 'Message 5';
SendSetup (0,addresslist);
```

(continued)

SendSetup (continued) SendSetup

```
For k := 1 to 5 do
   begin
      For i := 1 to 9 do
         message[i] := data[k,i];
      If k = 5 then eotmode := NLend;
      SendDataBytes (0,message,9,eotmode);
   end;
```

Turbo Pascal for Windows

```
var data         : array[1..5] of PChar;
   eotmode,i     : integer;
   addresslist   : addrlist;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := NOADDR;
eotmode := NULLend;
data[1] := 'Message 1';
data[2] := 'Message 2';
data[3] := 'Message 3';
data[4] := 'Message 4';
data[5] := 'Message 5';
SendSetup (0,addresslist);
For i := 1 to 5 do
   begin
      If i = 5 then eotmode := NLend;
      SendDataBytes (0,data[i]^,9,eotmode);
   end;
```

SetRWLS SetRWLS

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

Format: `SetRWLS (board,addresslist)`

`board` represents 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 type `addrlist`, as defined in the declaration files, and must be terminated by the value `NOADDR`.

Example:

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

```
var addresslist : addrlist;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := NOADDR;
SetRWLS (0,addresslist);
```

TestSRQ TestSRQ

Purpose: Determine the current state of the SRQ line.

Format: `TestSRQ (board,result)`

`board` represents 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 `0` into `result`.

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

Example:

Determine the current state of SRQ.

```
var result : integer;
TestSRQ (0,result);
If result := 1 then
   (* SRQ is asserted.     *)
else
   (* No SRQ at this time. *)
```

TestSys TestSys

Purpose: Cause devices to conduct self-tests.

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

`board` represents a board number. The GPIB devices whose addresses are contained in the `addresslist` 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. `addresslist` and `resultlist` are arrays of type `addrlist` as defined in the declaration files. `addresslist` must be terminated by the value `NOADDR`.

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.

Example:

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

```
var addresslist,resultlist : addrlist;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := NOADDR;
TestSys (0,addresslist,resultlist);
(* If any of the results are non-zero, the         *)
(* corresponding device has failed the test.       *)
```

Trigger Trigger

Purpose: Trigger a single device.

Format: `Trigger (board,address)`

`board` represents 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`, the Group Execute Trigger message is sent with no addressing and triggers 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`.

```
var address : integer;
address := 9 + 256 * 97;
Trigger (0,address);
```

TriggerList TriggerList

Purpose: Trigger multiple devices.

Format: `TriggerList (board,addresslist)`

`board` represents a board number. The GPIB devices whose addresses are contained in the address array are triggered simultaneously. The parameter `addresslist` is an array of type `addrlist`, defined in the declaration files. `addresslist` must be terminated by the value `NOADDR`. If the array contains only the value `NOADDR` or `NULL`, the Group Execute Trigger message is sent without addressing and triggers 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`.

```
var addresslist : addrlist;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := NOADDR;
TriggerList (0,addresslist);
```

WaitSRQ WaitSRQ

Purpose: Wait until a device asserts Service Request.

Format: `WaitSRQ (board,result)`

`board` represents 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. `WaitSRQ` suspends the program for, at most, the duration of the timeout period while 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.

```
var addresslist,resultlist : addrlist;
                    result : integer;
addresslist[0] := 8;
addresslist[1] := 9;
addresslist[2] := 10;
addresslist[3] := NOADDR;
WaitSRQ (0,result);
If result = 1 then
   AllSpoll (0,addresslist,resultlist);
(* resultlist now contains the serial poll         *)
(* responses for the three devices.                *)
```

NI-488.2 Example Programs

You can take full advantage of the ANSI/IEEE Standard 488.2-1987 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.

These examples illustrate the programming steps that you can follow to program a representative IEEE-488.2 instrument from your personal computer using the NI-488.2 routines. The applications are written in IBM/MS Pascal, QuickPascal, Turbo Pascal, and Turbo Pascal for Windows. 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 following steps explain how to use the driver to execute NI-488.2 programming and control sequences, without explaining how to determine those sequences.

1. Load the definitions of the NI-488.2 routines from the header files that are 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.

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

The NI-488.2 driver supports four interface boards. These boards are referenced by number from your application program. The reference number is `0` for the first board, `1` for the second board, and so on. 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 and identify the board by its base address. Refer to Chapter 2 of the *NI-488.2 Software Reference Manual for MS-DOS* for additional information about running and using `ibconf`. For NI-488.2 for Windows users, the configuration utility is named `wibconf`.

IBM/MS Pascal Program – NI-488.2 Routines

```
PROGRAM PSAMP488(input,output);

{$INCLUDE: 'decl.pas'}

type
     string30 = lstring(30);

var
  cmd           : cbuf;        (* Array of commands. cbuf is  *)
                               (* defined in DECL.PAS as a    *)
                               (* character array.            *)
  reading       : cbuf;        (* Data received from the      *)
                               (* Fluke 45.                   *)
  buffer        : string30;    (* Assigned the value of rd.   *)
                               (* Will be converted to        *)
                               (* numeric.                    *)
  sendstr       : string30;    (* GPIB command string.        *)
  instruments   : AddrList;    (* Array of primary addresses. *)
                               (* AddrList is defined in      *)
                               (* DECL.PAS as an integer      *)
                               (* array.                      *)
  result        : AddrList;    (* Array of listen addresses.  *)
  fluke         : integer;     (* Primary address of the      *)
                               (* Fluke 45.                   *)
  statusByte    : integer;     (* Serial poll response byte.  *)
  m, i          : integer;     (* FOR loop counters.          *)
  SRQasserted   : integer;     (* Set to indicate if SRQ      *)
                               (* asserted.                   *)
  num_listeners : integer;     (* Number of Listeners on      *)
                               (* GPIB.                       *)
  pad           : byte;        (* Primary address of          *)
                               (* Listener.                   *)
  num           : real4;       (* Numeric conversion of       *)
                               (* Reading.                    *)
  sum           : real4;       (* Accumulator of              *)
                               (* measurements.               *)

(* ============================================================
 *                        Procedure gpiberr
 *  The gpiberr procedure notifies you that an NI-488.2 routine
 *  failed by printing an error message.  The status variable
 *  ibsta is printed in decimal along with the mnemonic meaning
 *  of the bit position.   The status variable iberr is printed
 *  in decimal along with the mnemonic meaning of the decimal
 *  value.  The status variable ibcnt is printed in decimal.
 *
 *  The NI-488 function ibonl is called to disable the hardware
 *  and software.
 * ============================================================  *)
procedure gpiberr(msg:string30);
begin
```

```
    writeln(msg);

    write('ibsta = ', ibsta,' <');
    if ibsta and ERR   <>  0 then write (' ERR');
    if ibsta and TIMO  <>  0 then write (' TIMO');
    if ibsta and EEND  <>  0 then write (' END');
    if ibsta and SRQI  <>  0 then write (' SRQI');
    if ibsta and RQS   <>  0 then write (' RQS');
    if ibsta and CMPL  <>  0 then write (' CMPL');
    if ibsta and LOK   <>  0 then write (' LOK');
    if ibsta and REM   <>  0 then write (' REM');
    if ibsta and CIC   <>  0 then write (' CIC');
    if ibsta and ATN   <>  0 then write (' ATN');
    if ibsta and TACS  <>  0 then write (' TACS');
    if ibsta and LACS  <>  0 then write (' LACS');
    if ibsta and DTAS  <>  0 then write (' DTAS');
    if ibsta and DCAS  <>  0 then write (' DCAS');
    writeln(' >');

    write('iberr = ', iberr);
    if iberr = EDVR then writeln (' EDVR <DOS Error>');
    if iberr = ECIC then writeln (' ECIC <Not CIC>');
    if iberr = ENOL then writeln (' ENOL <No Listener>');
    if iberr = EADR then writeln (' EADR <Address error>');
    if iberr = EARG then writeln (' EARG <Invalid argument>');
    if iberr = ESAC then writeln (' ESAC <Not Sys Ctrlr>');
    if iberr = EABO then writeln (' EABO <Op. aborted>');
    if iberr = ENEB then writeln (' ENEB <No GPIB board>');
    if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
    if iberr = ECAP then writeln (' ECAP <No capability>');
    if iberr = EFSO then writeln (' EFSO <File sys. error>');
    if iberr = EBUS then writeln (' EBUS <Command error>');
    if iberr = ESTB then writeln (' ESTB <Status byte lost>');
    if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
    if iberr = ETAB then writeln (' ETAB <Table Overflow>');

    writeln('ibcnt = ', ibcnt);

(*  Call the ibonl function to disable the hardware and       *)
(*  software.                                                 *)

    ibonl(0,0);

end;

(* ============================================================
 *                      Procedure Found
 *  The Found procedure is called if the Fluke 45 has been
 *  identified as a Listener in the array result.
 *  The variable fluke is the primary address of the
 *  Fluke 45.  Ten measurements are read from the Fluke 45
 *  and the average of the sum is calculated.
 *
 *  The return statement terminates this procedure.
 * ============================================================ *)
procedure Found(fluke:integer);
```

```
begin

(*  Reset the Fluke 45 using the routines DevClear and Send. *)

(*  DevClear sends the GPIB Selected Device Clear (SDC)      *)
(*  message to the Fluke 45.  If the error bit (ERR) is set  *)
(*  in ibsta, call gpiberr with an error message.            *)

    DevClear(0, fluke);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('DevClear Error');
        return;
      end;

(*  Use the routine Send to send the NI-488.2 Reset command  *)
(*  to the Fluke 45.  Copy the string into the command array.*)
(*  The constant NLend, defined in DECL.PAS, instructs 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.               *)

    sendstr := '*RST';
    for i := 1 to 4 do
       cmd[i] := sendstr[i];
    Send(0, fluke, cmd, 4, NLend);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Send *RST Error');
        return;
      end;

(*  Use the routine 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 signal line, SRQ, when the measurement   *)
(*  has been completed and the Fluke 45 is ready to send the *)
(*  result (*SRE 16). Copy the string into the command array.*)
(*  If the error bit (ERR) is set, call gpiberr with an      *)
(*  error message.                                           *)

    sendstr := 'VAC; AUTO; TRIGGER 2; *SRE 16';
    for i := 1 to 29 do
       cmd[i] := sendstr[i];
    Send(0, fluke, cmd, 29, NLend);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Send Setup Error');
        return;
      end;

(*  Initialize the accumulator of the 10 measurements to     *)
(*  zero.                                                    *)

    sum := 0.0;
```

```
(*  Establish a FOR loop to read the 10 measurements.        *)
(*  The variable m serves as the counter for the FOR loop.   *)

    for m := 1 to 10 do
       begin

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

               sendstr := '*TRG; VAL1?';
               for i := 1 to 11 do
                  cmd[i] := sendstr[i];
               Send(0, fluke, cmd, 11, NLend);
               if (ibsta and ERR) <> 0 then
                 begin
                   gpiberr('Send Trigger Error');
                   return;
                 end;

          (*  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.              *)

              WaitSRQ(0, SRQasserted);
              if SRQasserted = 0 then
                begin
                  gpiberr('WaitSRQ Error');
                  return;
                end;

          (*  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.             *)

              ReadStatusByte(0, fluke, statusByte);
              if (ibsta and ERR) <> 0 then
                begin
                  gpiberr('ReadStatusByte Error');
                  return;
                end;

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

              if (statusByte and 16#10) <> 16#10 then
                begin
                  gpiberr('Improper Status Byte');
                  writeln('Status byte: ', statusByte);
                  return;
                end;
```

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

               Receive (0, fluke, Reading, 10, STOPend);
               if (ibsta and ERR) <> 0 then
                 begin
                   gpiberr('Receive Error');
                   return;
                 end;

          (*  Assign the array Reading to the string buffer.  *)
          (*  Remove spaces in buffer.  Print the measurement. *)

               buffer.len := lobyte(ibcnt) - 1;
               for i := 1 to (ibcnt - 1) do
                 buffer[i] := reading[i];
               writeln('Reading: ', buffer);
               writeln;

          (*  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 the conversion, add the value to   *)
          (*  the accumulator.                                *)

               if decode(buffer,num) then
                 sum := sum + num
               else
                 return;

     end; (*  Continue the FOR loop until 10 measurements     *)
          (*  are read.                                       *)

(*  Print the average of the 10 readings.                     *)

     writeln('The average of the 10 readings is ', sum/10);

end;

(* ============================================================
 *                            FindFluke
 *
 *  The FindFluke procedure is called from the MAIN body of
 *  this program to determine if the Fluke 45 is a Listener on
 *  the GPIB.
 *
 *  The return statement terminates this procedure.
 * ============================================================ *)
procedure FindFluke;
```

```
begin

(*  Send the *IDN? command to each device that was found.    *)
(*  Your board is at address 0 by default.  Your board does  *)
(*  not respond to *IDN?, so skip it.  Load the command      *)
(*  into the command array.                                  *)

    cmd[1] := '*';
    cmd[2] := 'I';
    cmd[3] := 'D';
    cmd[4] := 'N';
    cmd[5] := '?';

(*  Establish a FOR loop to determine if the Fluke 45 is a   *)
(*  Listener on the GPIB.  The variable i serves as the      *)
(*  counter of the FOR loop and as the index to the array    *)
(*  result.                                                  *)

    for i := 1 to num_listeners do
      begin

          (*  Send the identification query to each listen   *)
          (*  address in the array result.  The constant     *)
          (*  NLend, defined in DECL.PAS, instructs the      *)
          (*  routine 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.                 *)

             Send(0, result[i], cmd, 5, NLend);
             if (ibsta and ERR) <> 0 then
               begin
                 gpiberr('Send Error');
                 return;
               end;

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

             Receive(0, result[i], Reading, 10, STOPend);
             if (ibsta and ERR) <> 0 then
               begin
                 gpiberr('Receive Error');
                 return;
               end;

          (*  The low byte of the listen address is the      *)
          (*  primary address.  Assign the primary address   *)
          (*  of this device to the variable pad.            *)

             pad := lobyte(result[i]);
```

```
          (*  Assign the array Reading to the string buffer.  *)
          (*  Print the primary address and the name          *)
          (*  identification of the device.                   *)

             buffer.len := lobyte(ibcnt);
             for m := 1 to ibcnt do
               buffer[m] := reading[m];
             write('The instrument at address ', pad);
             writeln('is: ', buffer);

          (*  Determine if the name identification is the     *)
          (*  Fluke 45.  If it is the Fluke 45, assign the    *)
          (*  primary address to fluke, print a message that  *)
          (*  the Fluke 45 has been found, call the procedure*)
          (*  Found, return to the MAIN body of the program.  *)

             if buffer = 'FLUKE, 45,' then
               begin
                 fluke := result[i];
                 writeln('**** We found the Fluke 45 ****');
                 Found(fluke);
                 return;
               end;

      end;  (* End of FOR loop *)

(* Print a message that the Fluke 45 was not found.          *)

   writeln('Did not find the Fluke!')

end;

(* ============================================================
 *                              MAIN
 * ============================================================ *)

BEGIN

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

    SendIFC(0);
    if (ibsta and ERR) <> 0 THEN
      begin
        gpiberr('SendIFC Error');
        return;
      end;

(*  Create an array containing all valid GPIB primary        *)
(*  addresses. This array (instruments) is given to the      *)
(*  routine FindLstn to find all Listeners.  The constant    *)
(*  NOADDR, defined in DECL.PAS, signifies the end of the    *)
(*   array.                                                  *)

    for i := 0 to 30 do
       instruments[i] := i;

    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.                      *)

    writeln('Finding all listeners on the bus...');
    writeln;

    FindLstn (0, instruments, result, 31);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('FindLstn Error');
        return;
      end;

(*  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.                                         *)

    num_listeners := ibcnt - 1;

    writeln('No. of instruments found = ', num_listeners);

(*  Call the procedure FindFluke to determine if the Fluke   *)
(*  45 is a Listener on the GPIB.                            *)

    FindFluke;

(*  Call the ibonl function to disable the hardware and      *)
(*  software.                                                *)

    ibonl(0,0);

END.
```

QuickPascal Program – NI-488.2 Routines

```
PROGRAM QSAMP488(input,output);

{$I qpdecl.pas}

const
  maxlen = 10;              (* Maximum length of data array.  *)

var
  cmd          : cbuf;      (* Array of commands.  cbuf is    *)
                            (* defined in QPDECL.PAS as a     *)
                            (* character array.               *)
  reading      : cbuf;      (* Data received from the Fluke   *)
                            (* 45.                            *)
  buffer  : string[maxlen];(* Assigned the value of rd.      *)
                            (* Will be converted to numeric.  *)
  sendstr      : string[30];(* GPIB command string.           *)
  instruments: AddrList;    (* Array of primary addresses.    *)
                            (* AddrList is defined in         *)
                            (* QPDECL.PAS as an integer array.*)
  result       : AddrList;  (* Array of listen addresses.     *)
  fluke        : integer;   (* Primary address of the Fluke   *)
                            (* 45.                            *)
  statusByte   : integer;   (* Serial poll response byte.     *)
  m, i         : integer;   (* FOR loop counters.             *)
  SRQasserted  : integer;   (* Set to indicate if SRQ         *)
                            (* asserted.                      *)
  num_listeners : integer; (* Number of Listeners on GPIB.   *)
  pad          : integer;   (* Primary address of Listener.   *)
  code         : integer;   (* Procedure VAL parameter. VAL   *)
                            (* is Turbo Pascal conversion     *)
                            (* procedure.                     *)
  num          : real;      (* Numeric conversion of Reading. *)
  sum          : real;      (* Accumulator of measurements.   *)

(* ============================================================
 *                        Procedure gpiberr
 *  The gpiberr procedure notifies you that an NI-488.2
 *  routine failed by printing an error message.  The status
 *  variable ibsta is printed in decimal along with the
 *  mnemonic meaning of the bit position.  The status variable
 *  iberr is printed in decimal along with the mnemonic
 *  meaning of the decimal value.  The status variable ibcntl
 *  is printed in decimal.
 *
 *  The NI-488 function ibonl is called to disable the hardware
 *  and software.
 * ============================================================ *)
procedure gpiberr(msg:string);
```

```
begin

    writeln(msg);

    write('ibsta = ', ibsta,' <');
    if ibsta and ERR    <>  0 then write (' ERR');
    if ibsta and TIMO   <>  0 then write (' TIMO');
    if ibsta and EEND   <>  0 then write (' END');
    if ibsta and SRQI   <>  0 then write (' SRQI');
    if ibsta and RQS    <>  0 then write (' RQS');
    if ibsta and CMPL   <>  0 then write (' CMPL');
    if ibsta and LOK    <>  0 then write (' LOK');
    if ibsta and REM    <>  0 then write (' REM');
    if ibsta and CIC    <>  0 then write (' CIC');
    if ibsta and ATN    <>  0 then write (' ATN');
    if ibsta and TACS   <>  0 then write (' TACS');
    if ibsta and LACS   <>  0 then write (' LACS');
    if ibsta and DTAS   <>  0 then write (' DTAS');
    if ibsta and DCAS   <>  0 then write (' DCAS');
    writeln(' >');

    write('iberr = ', iberr);
    if iberr = EDVR then writeln (' EDVR <DOS Error>');
    if iberr = ECIC then writeln (' ECIC <Not CIC>');
    if iberr = ENOL then writeln (' ENOL <No Listener>');
    if iberr = EADR then writeln (' EADR <Address error>');
    if iberr = EARG then writeln (' EARG <Invalid argument>');
    if iberr = ESAC then writeln (' ESAC <Not Sys Ctrlr>');
    if iberr = EABO then writeln (' EABO <Op. aborted>');
    if iberr = ENEB then writeln (' ENEB <No GPIB board>');
    if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
    if iberr = ECAP then writeln (' ECAP <No capability>');
    if iberr = EFSO then writeln (' EFSO <File sys. error>');
    if iberr = EBUS then writeln (' EBUS <Command error>');
    if iberr = ESTB then writeln (' ESTB <Status byte lost>');
    if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
    if iberr = ETAB then writeln (' ETAB <Table Overflow>');

    writeln('ibcnt = ', ibcntl);

(*  Call the ibonl function to disable the hardware and       *)
(*  software.                                                 *)

    ibonl(0,0);

end;

(* ============================================================
 *                        Procedure Found
 *  The Found procedure is called if the Fluke 45 has been
 *  identified as a Listener in the array result.  The variable
 *  fluke is the primary address of the Fluke 45.  Ten
 *  measurements are read from the Fluke 45 and the average
 *  of the sum is calculated.
 *
 *  The halt procedure stops execution of this program.
 *
 *  The exit procedure terminates this procedure.
 * ============================================================ *)
procedure Found(fluke:integer);
```

```
begin

(*  Reset the Fluke 45 using the routines DevClear and Send.  *)

(*  DevClear sends the GPIB Selected Device Clear (SDC)      *)
(*  message to the Fluke 45.  If the error bit (ERR) is set  *)
(*  in ibsta, call gpiberr with an error message.            *)

    DevClear(0, fluke);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('DevClear Error');
        halt;
      end;

(*  Use the routine Send to send the NI-488.2 Reset command  *)
(*  to the Fluke 45.  Copy the string into the command array.*)
(*  The constant NLend, defined in QPDECL.PAS, instructs     *)
(*  Send to append a linefeed character with EOI asserted to *)
(*  the end of the message.  If the error bit (ERR) is set   *)
(*  in set inibsta, call gpiberr with an error message.      *)

    sendstr := '*RST';
    for i := 1 to 4 do
       cmd[i] := sendstr[i];
    Send(0, fluke, cmd, 4, NLend);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Send *RST Error');
        halt;
      end;

(*  Use the routine 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 signal line, SRQ, when the measurement   *)
(*  has been completed and the Fluke 45 is ready to send     *)
(*  the result (*SRE 16).  Copy the string into the command  *)
(*  array. If the error bit (ERR) is set, call gpiberr       *)
(*  with an error message.                                   *)

    sendstr := 'VAC; AUTO; TRIGGER 2; *SRE 16';
    for i := 1 to 29 do
       cmd[i] := sendstr[i];
    Send(0, fluke, cmd, 29, NLend);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Send Setup Error');
        halt;
      end;

(*  Initialize the accumulator of the 10 measurements to     *)
(*  zero.                                                    *)

    sum := 0.0;
```

```
(*  Establish a FOR loop to read the 10 measurements.        *)
(*  The variable m serves as the counter for the FOR loop.   *)

    for m := 1 to 10 do
       begin

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

              sendstr := '*TRG; VAL1?';
              for i := 1 to 11 do
                 cmd[i] := sendstr[i];
              Send(0, fluke, cmd, 11, NLend);
              if (ibsta and ERR) <> 0 then
                begin
                  gpiberr('Send Trigger Error');
                  halt;
                end;

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

              WaitSRQ(0, SRQasserted);
              if SRQasserted = 0 then
                begin
                  gpiberr('WaitSRQ Error');
                  halt;
                end;

          (*  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.             *)

               ReadStatusByte(0, fluke, statusByte);
               if (ibsta and ERR) <> 0 then
                 begin
                   gpiberr('ReadStatusByte Error');
                   halt;
                 end;

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

               if (statusByte and $10) <> $10 then
                 begin
                   gpiberr('Improper Status Byte');
                   writeln('Status byte: ', statusByte);
                   halt;
                 end;
```

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

               Receive (0, fluke, Reading, 10, STOPend);
               if (ibsta and ERR) <> 0 then
                 begin
                   gpiberr('Receive Error');
                   halt;
                 end;

          (*  Assign the array Reading to the string buffer.   *)
          (*  Remove spaces in buffer.  Print the measurement. *)

               for i:= 1 to (ibcnt - 1) do
                 buffer[i] := reading[i];
               buffer[0] := chr(ibcnt - 1);
               delete(buffer, ibcnt, maxlen - ibcnt + 1);
               writeln('Reading: ', buffer);
               writeln;

          (*  Convert the measurement to its numeric value.   *)
          (*  If there is an error during the conversion,     *)
          (*  print the index of the character that did not   *)
          (*  convert and terminate this program.  If an      *)
          (*  error does not occur during the conversion, add *)
          (*  the value to the accumulator.                   *)

              val(buffer, num, code);
              if code <> 0 then
                begin
                  writeln('Error at position: ', code);
                  exit;
                end
              else
                sum := sum + num;

    end;   (*  Continue the FOR loop until 10 measurements    *)
           (*  are read.                                      *)

(*  Print the average of the 10 readings.                     *)

    writeln('The average of the 10 readings is ', sum/10);

end;

(* ============================================================
 *                            FindFluke
 *  The FindFluke procedure is called from the MAIN body of
 *  this program to determine if the Fluke 45 is a Listener
 *  on the GPIB.
 *
 *  The halt procedure stops execution of this program.
 *
 *  The exit procedure terminates this procedure.
 * ============================================================*)
procedure FindFluke;
```

```
begin

(*  Send the *IDN? command to each device that was found.    *)
(*  Your board is at address 0 by default.  Your board does  *)
(*  not respond to *IDN?, so skip it.  Load the command      *)
(*  into  the command array.                                 *)

    cmd[1] := '*';
    cmd[2] := 'I';
    cmd[3] := 'D';
    cmd[4] := 'N';
    cmd[5] := '?';

(*  Establish a FOR loop to determine if the Fluke 45 is a   *)
(*  Listener on the GPIB.  The variable i serves as the      *)
(*  counter of the FOR loop and as the index to the array    *)
(*  result.                                                  *)

    for i := 1 to num_listeners do
      begin

          (*  Send the identification query to each listen   *)
          (*  address in the array result.  The constant     *)
          (*  NLend, defined in QPDECL.PAS, instructs the    *)
          (*  routine 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.                 *)

              Send(0, result[i], cmd, 5, NLend);
              if (ibsta and ERR) <> 0 then
                begin
                  gpiberr('Send Error');
                  halt;
                end;

          (*  Read the name identification response returned *)
          (*  from each device.  Store the response in the   *)
          (*  array Reading.  The constant STOPend, declared *)
          (*  in QPDECL.PAS, instructs the routine Receive to*)
          (*  terminate the read when END is detected.  If   *)
          (*  the error bit (ERR) is set in ibsta, call      *)
          (*  gpiberr with an error message.                 *)

              Receive(0, result[i], Reading, 10, STOPend);
              if (ibsta and ERR) <> 0 then
                begin
                  gpiberr('Receive Error');
                  halt;
                end;

          (*  The low byte of the listen address is the      *)
          (*  primary address.  Assign the primary address   *)
          (*  of this device to the variable pad.            *)

            pad := Lo(result[i]);
```

```
          (*  Assign the array Reading to the string buffer.  *)
          (*  Print the primary address and the name          *)
          (*  identification of the device.                   *)

             buffer := reading;
             write('The instrument at address ', pad );
             writeln(' is: ', buffer);

          (*  Determine if the name identification is the     *)
          (*  Fluke 45.  If it is the Fluke 45, assign pad to *)
          (*  fluke, print a message that the Fluke 45 has    *)
          (*  been found, call the procedure Found, and       *)
          (*  return to the MAIN body of the program.         *)

               if buffer = 'FLUKE, 45,' then
                 begin
                  fluke := pad;
                   writeln('**** We found the Fluke 45 ****');
                   Found(fluke);
                   exit;
                 end;

      end;  (* End of FOR loop *)

(* Print a message that the Fluke 45 was not found.          *)

   writeln('Did not find the Fluke!')

end;

(* ============================================================
 *                              MAIN
 * ============================================================*)

BEGIN

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

    SendIFC(0);
    if (ibsta and ERR) <> 0 THEN
      begin
        gpiberr('SendIFC Error');
        halt;
      end;

(*  Create an array containing all valid GPIB primary        *)
(*  addresses. This array (instruments) will be given to     *)
(*  the  routine FindLstn to find all Listeners.             *)
(*  The constant  NOADDR, defined in QPDECL.PAS, signifies   *)
(*  the end of the array.                                    *)

    for i := 0 to 30 do
       instruments[i] := i;

    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.                      *)

    writeln('Finding all listeners on the bus...');
    writeln;

    FindLstn (0, instruments, result, 31);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('FindLstn Error');
        halt;
      end;

(*  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.                                      *)

    num_listeners := ibcnt - 1;

    writeln('No. of instruments found = ', num_listeners);

(*  Call the procedure FindFluke to determine if the         *)
(*  Fluke 45 is a Listener on the GPIB.                      *)

    FindFluke;

(*  Call the ibonl function to disable the hardware and      *)
(*  software.                                                *)

    ibonl(0,0);

END.
```

Turbo Pascal Program – NI-488.2 Routines

```
PROGRAM TSAMP488(input,output);

uses tpdecl;

const
  maxlen = 10;              (* Maximum length of data array.  *)
var
  cmd         : cbuf;       (* Array of commands.  cbuf is    *)
                            (* defined in TPDECL.PAS as a     *)
                            (* character array.               *)
  reading     : cbuf;       (* Data received from the Fluke   *)
                            (* 45.                            *)
  buffer  : string[maxlen];(* Assigned the value of rd.       *)
                            (* Will be converted to numeric.  *)
  sendstr     : string[30];(* GPIB command string.            *)
  instruments : AddrList;   (* Array of primary addresses.    *)
                            (* AddrList is defined in         *)
                            (* TPDECL.PAS as an integer array.*)
  result      : AddrList;   (* Array of listen addresses.     *)
  fluke       : integer;    (* Primary address of the Fluke   *)
                            (* 45.                            *)
  statusByte  : integer;    (* Serial poll response byte.     *)
  m, i        : integer;    (* FOR loop counters.             *)
  SRQasserted : integer;    (* Set to indicate if SRQ         *)
                            (* asserted.                      *)
  num_listeners: integer;   (* Number of Listeners on GPIB.   *)
  pad         : integer;    (* Primary address of Listener.   *)
  code        : integer;    (* Procedure VAL parameter.  VAL  *)
                            (* is Turbo Pascal conversion     *)
                            (* procedure.                     *)
  num         : real;       (* Numeric conversion of Reading. *)
  sum         : real;       (* Accumulator of measurements.   *)

(* ============================================================
 *                      Procedure gpiberr
 *  The gpiberr procedure notifies you that an NI-488.2 routine
 *  failed by printing an error message.  The status variable
 *  ibsta is printed in decimal along with the mnemonic meaning
 *  of the bit position.   The status variable iberr is printed
 *  in decimal along with the mnemonic meaning of the decimal
 *  value.  The status variable ibcnt is printed in decimal.
 *
 *  The NI-488 function ibonl is called to disable the hardware
 *  and software.
 * ============================================================*)
procedure gpiberr(msg:string);
```

```
begin

    writeln(msg);

    write('ibsta = ', ibsta,' <');
    if ibsta and ERR    <>  0 then write (' ERR');
    if ibsta and TIMO   <>  0 then write (' TIMO');
    if ibsta and EEND   <>  0 then write (' END');
    if ibsta and SRQI   <>  0 then write (' SRQI');
    if ibsta and RQS    <>  0 then write (' RQS');
    if ibsta and CMPL   <>  0 then write (' CMPL');
    if ibsta and LOK    <>  0 then write (' LOK');
    if ibsta and REM    <>  0 then write (' REM');
    if ibsta and CIC    <>  0 then write (' CIC');
    if ibsta and ATN    <>  0 then write (' ATN');
    if ibsta and TACS   <>  0 then write (' TACS');
    if ibsta and LACS   <>  0 then write (' LACS');
    if ibsta and DTAS   <>  0 then write (' DTAS');
    if ibsta and DCAS   <>  0 then write (' DCAS');
    writeln(' >');

    write('iberr = ', iberr);
    if iberr = EDVR then writeln (' EDVR <DOS Error>');
    if iberr = ECIC then writeln (' ECIC <Not CIC>');
    if iberr = ENOL then writeln (' ENOL <No Listener>');
    if iberr = EADR then writeln (' EADR <Address error>');
    if iberr = EARG then writeln (' EARG <Invalid argument>');
    if iberr = ESAC then writeln (' ESAC <Not Sys Ctrlr>');
    if iberr = EABO then writeln (' EABO <Op. aborted>');
    if iberr = ENEB then writeln (' ENEB <No GPIB board>');
    if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
    if iberr = ECAP then writeln (' ECAP <No capability>');
    if iberr = EFSO then writeln (' EFSO <File sys. error>');
    if iberr = EBUS then writeln (' EBUS <Command error>');
    if iberr = ESTB then writeln (' ESTB <Status byte lost>');
    if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
    if iberr = ETAB then writeln (' ETAB <Table Overflow>');

    writeln('ibcnt = ', ibcntl);

(*  Call the ibonl function to disable the hardware and      *)
(*  software.                                                *)

    ibonl(0,0);

end;

(* ============================================================
 *                        Procedure Found
 *  The Found procedure is called if the Fluke 45 has been
 *  identified as a Listener in the array result.  The variable
 *  fluke is the primary address of the Fluke 45.  Ten
 *  measurements are read from the Fluke 45 and the average of
 *  the sum is calculated.
 *
 *  The return statement terminates this procedure.
 * ============================================================*)
procedure Found(fluke:integer);
```

```
begin

(*  Reset the Fluke 45 using the routines DevClear and Send. *)

(*  DevClear sends the GPIB Selected Device Clear (SDC)      *)
(*  message to the Fluke 45.  If the error bit (ERR) is set  *)
(*  in ibsta, call gpiberr with an error message.            *)

    DevClear(0, fluke);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('DevClear Error');
        halt;
      end;

(*  Use the routine Send to send the NI-488.2 Reset command  *)
(*  to the Fluke 45.  Copy the string into the command array.*)
(*  The constant NLend, defined TPDECL.PAS, instructs 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.                *)

    sendstr := '*RST';
    for i := 1 to 4 do
       cmd[i] := sendstr[i];
    Send(0, fluke, cmd, 4, NLend);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Send *RST Error');
        halt;
      end;

(*  Use the routine 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 signal line, SRQ,               *)
(*  when the measurement has been completed and the Fluke    *)
(*  45 is ready to send the result (*SRE 16).  Copy the      *)
(*  string into the command array.  If the error bit (ERR)   *)
(*  is set, call gpiberr with an error message.              *)

    sendstr := 'VAC; AUTO; TRIGGER 2; *SRE 16';
    for i := 1 to 29 do
       cmd[i] := sendstr[i];
    Send(0, fluke, cmd, 29, NLend);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Send Setup Error');
        halt;
      end;

(*  Initialize the accumulator of the 10 measurements to     *)
(*  zero.                                                    *)

    sum := 0.0;
```

```
(*  Establish a FOR loop to read the 10 measurements.  The   *)
(*  variable m serves as the counter for the FOR loop.       *)

    for m := 1 to 10 do
       begin

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

               sendstr := '*TRG; VAL1?';
               for i := 1 to 11 do
                  cmd[i] := sendstr[i];
               Send(0, fluke, cmd, 11, NLend);
               if (ibsta and ERR) <> 0 then
                 begin
                   gpiberr('Send Trigger Error');
                   halt;
                 end;

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

               WaitSRQ(0, SRQasserted);
               if SRQasserted = 0 then
                 begin
                   gpiberr('WaitSRQ Error');
                   halt;
                 end;

          (*  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.             *)

               ReadStatusByte(0, fluke, statusByte);
               if (ibsta and ERR) <> 0 then
                 begin
                   gpiberr('ReadStatusByte Error');
                   halt;
                 end;

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

               if (statusByte and $10) <> $10 then
                 begin
                   gpiberr('Improper Status Byte');
                   writeln('Status byte: ', statusByte);
                   halt;
                 end;
```

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

               Receive (0, fluke, Reading, 10, STOPend);
               if (ibsta and ERR) <> 0 then
                 begin
                   gpiberr('Receive Error');
                   halt;
                 end;

          (*  Assign the array Reading to the string buffer.  *)
          (*  Remove spaces in buffer.  Print the measurement.*)

               for i:= 1 to (ibcnt - 1) do
                 buffer[i] := reading[i];
               buffer[0] := chr(ibcnt - 1);
               writeln('Reading: ', buffer);
               writeln;

          (*  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 the conversion, add the value to  *)
          (*  the accumulator.                               *)

               val(buffer, num, code);
               if code <> 0 then
                 begin
                   writeln('Error at position: ', code);
                   exit;
                 end
               else
                 sum := sum + num;

     end;  (*  Continue the FOR loop until 10 measurements    *)
           (*  are read.                                      *)

(*  Print the average of the 10 readings.                     *)

     writeln('The average of the 10 readings is ', sum/10);

end;

(* ============================================================
 *                          FindFluke
 *  The FindFluke procedure is called from the MAIN body of
 *  this program to determine if the Fluke 45 is a Listener
 *  on the GPIB.
 *
 *  The halt procedure stops execution of this program.
 *
 *  The exit procedure terminates this procedure.
 * ============================================================ *)
procedure FindFluke;
```

```
begin

(*  Send the *IDN? command to each device that was found.    *)
(*  Your board is at address 0 by default.  Your board does  *)
(*  not respond to *IDN?, so skip it.  Load the command into *)
(*  the command array.                                       *)

    cmd[1] := '*';
    cmd[2] := 'I';
    cmd[3] := 'D';
    cmd[4] := 'N';
    cmd[5] := '?';

(*  Establish a FOR loop to determine if the Fluke 45 is a   *)
(*  Listener on the GPIB.  The variable i serves as the      *)
(*  counter of the FOR loop and as the index to the array    *)
(*  result.                                                  *)

    for i := 1 to num_listeners do
      begin

          (*  Send the identification query to each listen    *)
          (*  address in the array result.  The constant      *)
          (*  NLend, defined in TPDECL.PAS, instructs the     *)
          (*  routine 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.                  *)

             Send(0, result[i], cmd, 5, NLend);
             if (ibsta and ERR) <> 0 then
               begin
                 gpiberr('Send Error');
                 halt;
               end;

          (*  Read the name identification response returned  *)
          (*  from each device.  Store the response in the    *)
          (*  array Reading.  The constant STOPend, declared  *)
          (*  in TPDECL.PAS, instructs the routine Receive to *)
          (*  terminate the read when END is detected. If the *)
          (*  error bit is set in ibsta, call gpiberr with an  *)
          (*  error message.                                  *)

             Receive(0, result[i], Reading, 10, STOPend);
             if (ibsta and ERR) <> 0 then
               begin
                 gpiberr('Receive Error');
                 halt;
               end;

          (*  The low byte of the listen address is the       *)
          (*  primary address.  Assign the primary address of *)
          (*  this device to the variable pad.                *)

             pad := Lo(result[i]);
```

```
          (*  Assign the array Reading to the string buffer.  *)
          (*  Print the primary address and the name          *)
          (*  identification of the device.                   *)

             buffer := reading;
             write('The instrument at address ', pad);
             writeln(' is: ', buffer);

          (*  Determine if the name identification is the     *)
          (*  Fluke 45.  If it is the Fluke 45, assign the    *)
          (*  primary address to fluke, print a message that *)
          (*  the Fluke 45 has been found, call the procedure*)
          (*  Found, return to the MAIN body of the program.  *)

             if buffer = 'FLUKE, 45,' then
               begin
                 fluke := pad;
                 writeln('**** We found the Fluke 45 ****');
                 Found(fluke);
                 exit;
               end;

       end;  (* End of FOR loop *)

(* Print a message that the Fluke 45 was not found.          *)

   writeln('Did not find the Fluke!')

end;

(* ============================================================
 *                              MAIN
 * ============================================================*)

BEGIN

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

    SendIFC(0);
    if (ibsta and ERR) <> 0 THEN
      begin
        gpiberr('SendIFC Error');
        halt;
      end;

(*  Create an array containing all valid GPIB primary        *)
(*  addresses. This array (instruments) will be given to the *)
(*  routine FindLstn to find all Listeners.  The constant    *)
(*  NOADDR, defined in TPDECL.PAS, signifies the end of the  *)
(*  array.                                                   *)

    for i := 0 to 30 do
       instruments[i] := i;

    instruments[31] := NOADDR;
```

```
(*  Print a message to tell 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.               *)

    writeln('Finding all listeners on the bus...');
    writeln;

    FindLstn (0, instruments, result, 31);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('FindLstn Error');
        halt;
      end;

(*  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.                                         *)

    num_listeners := ibcnt - 1;

    writeln('No. of instruments found = ', num_listeners);

(*  Call the procedure FindFluke to determine if the         *)
(*  Fluke 45 is a Listener on the GPIB.                      *)

    FindFluke;

(*  Call the ibonl function to disable the hardware and      *)
(*  software.                                                *)

    ibonl(0,0);

END.
```

Turbo Pascal for Windows Program – NI-488.2 Routines

```
PROGRAM WSAMP488(input,output);

uses wincrt, tpwgpib;

const
 flukename : array [0..10] of char = 'FLUKE, 45,';

var
 cmd          : PChar;                (* Commands sent to      *)
                                      (* the Fluke 45.         *)
 reading      : array[0..10] of char; (* Data received from    *)
                                      (* the Fluke 45.         *)
 instruments  : AddrList;             (* Array of primary      *)
                                      (* addresses.            *)
                                      (* AddrList is defined   *)
                                      (* in TPWGPIB.PAS        *)
                                      (* as an integer array.  *)
 result       : AddrList;             (* Array of listen       *)
                                      (* addresses.            *)
 fluke        : integer;              (* Primary address of    *)
                                      (* the Fluke 45.         *)
 statusByte   : integer;              (* Serial poll response  *)
                                      (* byte.                 *)
 m, i         : integer;              (* FOR loop counters.    *)
 SRQasserted  : integer;              (* Set to indicate if    *)
                                      (* SRQ asserted.         *)
 num_listeners: integer;              (* Number of listeners   *)
                                      (* on GPIB.              *)
 pad          : integer;              (* Primary address of    *)
                                      (* listener.             *)
 code         : integer;              (* Procedure VAL         *)
                                      (* parameter.  VAL is    *)
                                      (* Turbo Pascal          *)
                                      (* conversion procedure. *)
 num          : real;                 (* Numeric conversion    *)
                                      (* of READING.           *)
 sum          : real;                 (* Accumulator of        *)
                                      (* measurements.         *)

(* ============================================================
 *                      Procedure GPIBERR
 *  This procedure will notify you that a NI-488.2 function
 *  failed by printing an error message.  The status variable
 *  IBSTA will be printed in decimal along with the mnemonic
 *  meaning of the bit position.  Thestatus variable IBERR will
 *  be printed in decimal along with the mnemonic meaning of
 *  the decimal value.  The status variable IBCNTL will be
 *  printed in decimal.
 *
 *  The NI-488 function IBONL is called to disable the hardware
 *  and software.
 * ============================================================*)
procedure gpiberr(msg:string);
```

```
begin

   writeln(msg);

   write('ibsta = ', ibsta,' <');
   if ibsta and ERR  <> 0 then write (' ERR');
   if ibsta and TIMO <> 0 then write (' TIMO');
   if ibsta and EEND <> 0 then write (' END');
   if ibsta and SRQI <> 0 then write (' SRQI');
   if ibsta and RQS  <> 0 then write (' RQS');
   if ibsta and CMPL <> 0 then write (' CMPL');
   if ibsta and LOK  <> 0 then write (' LOK');
   if ibsta and REM  <> 0 then write (' REM');
   if ibsta and CIC  <> 0 then write (' CIC');
   if ibsta and ATN  <> 0 then write (' ATN');
   if ibsta and TACS <> 0 then write (' TACS');
   if ibsta and LACS <> 0 then write (' LACS');
   if ibsta and DTAS <> 0 then write (' DTAS');
   if ibsta and DCAS <> 0 then write (' DCAS');
   writeln(' >');

   write('iberr = ', iberr);
   if iberr = EDVR then writeln (' EDVR <DOS Error>');
   if iberr = ECIC then writeln (' ECIC <Not CIC>');
   if iberr = ENOL then writeln (' ENOL <No Listener>');
   if iberr = EADR then writeln (' EADR <Address error>');
   if iberr = EARG then writeln (' EARG <Invalid argument>');
   if iberr = ESAC then writeln (' ESAC <Not Sys Ctrlr>');
   if iberr = EABO then writeln (' EABO <Op. aborted>');
   if iberr = ENEB then writeln (' ENEB <No GPIB board>');
   if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
   if iberr = ECAP then writeln (' ECAP <No capability>');
   if iberr = EFSO then writeln (' EFSO <File sys. error>');
   if iberr = EBUS then writeln (' EBUS <Command error>');
   if iberr = ESTB then writeln (' ESTB <Status byte lost>');
   if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
   if iberr = ETAB then writeln (' ETAB <Table Overflow>');

   writeln('ibcnt = ', ibcntl);

(*  Call the IBONL function to disable the hardware and       *)
(*  software.                                                 *)

   ibonl(0,0);

end;
```

```
(* ============================================================
 *                       Procedure FOUND
 *  This procedure is called if the Fluke 45 has been identified
 *  as a listener in the array RESULT.  The variable FLUKE is
 *  the primary address of the Fluke 45.  Ten measurements are
 *  read from the Fluke 45 and the average of the sum is
 *  calculated.
 *
 *  The HALT procedure stops execution of this program.
 *
 *  The EXIT procedure terminates this procedure.
 * ============================================================*)
procedure Found(fluke:integer);

begin

(*  Reset the Fluke 45 using the subroutines DevClear and     *)
(*  Send.                                                     *)

(*  DevClear will send the GPIB Selected Device Clear (SDC)   *)
(*  message to the  Fluke 45.  If the error bit ERR is set    *)
(*  in IBSTA, call GPIBERR with an error message.             *)

    DevClear(0, fluke);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('DevClear Error');
        halt;
      end;

(* Use the subroutine Send to send the 488.2 Reset command    *)
(* to the Fluke 45.  The constant NLend, defined TPWGPIB.PAS, *)
(* instructs 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.       *)

    cmd := '*RST';
    Send(0, fluke, cmd^, 4, NLend);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Send *RST Error');
        halt;
      end;

(*  Use the subroutine 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 signal 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, call      *)
(*  GPIBERR with an error message.                            *)
```

```
    cmd := 'VAC; AUTO; TRIGGER 2; *SRE 16';
    Send(0, fluke, cmd^, 29, NLend);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Send Setup Error');
        halt;
      end;

(*  Initialize the accumulator of the 10 measurements to      *)
(*  zero.                                                     *)

    sum := 0.0;

(*  Establish FOR loop to read the 10 measurements.  The      *)
(*  variable m will serve as the counter for the FOR loop.    *)

    for m := 1 to 10 do
       begin

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

              cmd := '*TRG; VAL1?';
              Send(0, fluke, cmd^, 11, NLend);
              if (ibsta and ERR) <> 0 then
                begin
                  gpiberr('Send Trigger Error');
                  halt;
                end;

          (*  Wait for the Fluke 45 to assert SRQ, meaning it  *)
          (*  is ready with the measurement.  If SRQ is not    *)
          (*  asserted within the timeout period, call GPIBERR *)
          (*  with an error message.  The timeout period by    *)
          (*  default is 10 seconds.                           *)

              WaitSRQ(0, SRQasserted);
              if SRQasserted = 0 then
                begin
                  gpiberr('WaitSRQ Error');
                  halt;
                end;

          (*  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.                   *)

              ReadStatusByte(0, fluke, statusByte);
              if (ibsta and ERR) <> 0 then
                begin
                  gpiberr('ReadStatusByte Error');
                  halt;
                end;
```

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

            if (statusByte and $10) <> $10 then
              begin
                gpiberr('Improper Status Byte');
                writeln('Status byte: ', statusByte);
                halt;
              end;

        (*  Read the Fluke 45 measurement.  Store            *)
        (*  measurement in the array READING.  The constant  *)
        (*  STOPend, defined in TPWGPIB.PAS, instructs the   *)
        (*  subroutine Receive to terminate the read when    *)
        (*  END is detected.  If the error bit ERR is set in *)
        (*  IBSTA, call the GPIBERR with an error message.   *)

            Receive (0, fluke, Reading, 10, STOPend);
            if (ibsta and ERR) <> 0 then
              begin
                gpiberr('Receive Error');
                halt;
              end;

        (*  Remove Spaces in READING. Print the measurement. *)

             reading[ibcnt-1] := #0;
             writeln('Reading: ', reading);
             writeln;

        (*  Convert the measurement to its numeric value.    *)
        (*  If there is an error during the conversion,      *)
        (*  print the index of the character that did not    *)
        (*  convert and terminate this program.  If an error *)
        (*  does not occur during the conversion, add the    *)
        (*  value to the accumulator.                        *)

            val(reading, num, code);
            if code <> 0 then
              begin
                writeln('Error at position: ', code);
                exit;
              end
            else
              sum := sum + num;

    end;  (*  Continue FOR loop until 10 measurements are read. *)

(*  Print the average of the 10 readings.                    *)

    writeln('The average of the 10 readings is ', sum/10);

end;
```

```
(* ============================================================
 *                    PROCEDURE FINDFLUKE
 *
 *  This procedure is called from the MAIN body of this program
 *  to determine if the Fluke 45 is a listener on the GPIB.
 *
 *  The HALT procedure stops execution of this program.
 *
 *  The EXIT procedure terminates this procedure.
 * ============================================================*)
procedure FindFluke;

begin

(*  Send the *IDN? command to each of the devices that was    *)
(*  found.  Your board is at address 0 by default.  Your      *)
(*  board does not respond to *IDN?, so skip it.              *)
(*  Establish FOR loop to determine if the Fluke 45 is a      *)
(*  listener on the GPIB.  The variable i will serve as the   *)
(*  counter of the FOR loop and as the index to the array     *)
(*  RESULT.                                                   *)

    for i := 1 to num_listeners do
      begin

        (*  Send the identification query to each listen      *)
        (*  address in the array RESULT.  The constant NLend, *)
        (*  defined in TPWGPIB.PAS, instructs the subroutine  *)
        (*  Send to append a linefeed character with EOI      *)
        (*  asserted to the end of the message.  If the error *)
        (*  bit is set in IBSTA, call GPIBERR with an error   *)
        (*  message.                                          *)

            cmd := '*IDN?';
            Send(0, result[i], cmd^, 5, NLend);
            if (ibsta and ERR) <> 0 then
              begin
                gpiberr('Send Error');
                halt;
              end;

        (*  Read the name identification response returned    *)
        (*  from each device.  Store response in the array    *)
        (*  READING.  The constant STOPend, declared in       *)
        (*  TPWGPIB.PAS, instructs the subroutine Receive to  *)
        (*  terminate the read when END is detected.  If the  *)
        (*  error bit is set in IBSTA, call GPIBERR with an   *)
        (*  error message.                                    *)

            Receive(0, result[i], Reading, 10, STOPend);
            if (ibsta and ERR) <> 0 then
              begin
                gpiberr('Receive Error');
                halt;
              end;
```

```
        (*  The low byte of the listen address is the primary *)
        (*  address.  Assign the variable PAD the primary     *)
        (*  address of this device.                           *)

            pad := Lo(result[i]);

        (*  Print the primary address and the name            *)
        (*  identification of the  device.                    *)

             writeln('The instrument at address ', pad ,
                     ' is: ', reading);

        (*  Determine if the name identification is the       *)
        (*  Fluke 45.  If it is the Fluke 45, assign PAD to   *)
        (*  FLUKE, print message that the Fluke 45 has been   *)
        (*  found, call the procedure FOUND, return to MAIN   *)
        (*  body of the program.                              *)

            if reading = flukename then
              begin
                fluke := pad;
                writeln('**** We found the Fluke 45 ****');
                Found(fluke);
                exit;
              end;

      end;  (* End of FOR loop *)

(* Print message the Fluke 45 was not found.                  *)

   writeln('Did not find the Fluke!')

end;

(* ============================================================
 *                          MAIN
 * ============================================================*)

BEGIN

(*  Your board needs to be the Controller-In-Charge in order  *)
(*  to find all listeners on the GPIB.  To accomplish this,   *)
(*  the subroutine SendIFC is called.  If the error bit ERR   *)
(*  is set in IBSTA, call GPIBERR with an error message.      *)

    SendIFC(0);
    if (ibsta and ERR) <> 0 THEN
      begin
        gpiberr('SendIFC Error');
        halt;
      end;
```

```
(*  Create an array containing all valid GPIB primary         *)
(*  addresses.  This array (INSTRUMENTS) will be given to     *)
(*  the subroutine FindLstn to find all listeners.  The       *)
(*  constant NOADDR, defined in TPWGPIB.PAS, signifies  the   *)
(*  end of the array.                                         *)

    for i := 0 to 30 do
       instruments[i] := i;
    instruments[31] := NOADDR;

(*  Print message to tell 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.                                         *)

    writeln('Finding all listeners on the bus...');
    writeln;

    FindLstn (0, instruments, result, 31);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('FindLstn Error');
        halt;
      end;

(*  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.                                          *)

    num_listeners := ibcnt - 1;

    writeln('No. of instruments found = ', num_listeners);

(*  Call the procedure FINDFLUKE to determine if the Fluke 45 *)
(*  is a listener on the GPIB.                                *)

    FindFluke;

(*  Call the IBONL function to disable the hardware and       *)
(*  software.                                                 *)

    ibonl(0,0);

END.
```

Chapter 3
NI-488 Function Descriptions

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

IBASK IBASK

Purpose: Return information about software configuration parameters.

Format: `ibask (ud,option,value)`

`ud` represents a GPIB interface board or a device. The selected configuration item is returned in the integer specified by value. Table 3-1 and Table 3-2 list the valid configurations parameter options for `ibask`.

Table 3-1. ibask Board Configuration Parameter Options

Options (Constants)	Options (Values)	Returned Information
IbaPAD	$0001	The current primary address of the board. See `ibpad`.
IbaSAD	$0002	The current secondary address of the board. See `ibsad`.
IbaTMO	$0003	The current I/O timeout of the board. See `ibtmo`.
IbaEOT	$0004	zero = The GPIB EOI line is not asserted at the end of a write operation. non-zero = EOI is asserted at the end of a write. See `ibeot`.
IbaPPC	$0005	The current parallel poll configuration information of the board. See `ibppc`.

(continues)

IBASK (continued) IBASK

Table 3-1. ibask Board Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Returned Information
IbaAUTOPOLL	$0007	zero = Automatic serial polling is disabled. non-zero = Automatic serial polling is enabled. Refer to the NI-488.2 user manual for more information about automatic serial polling.
IbaCICPROT	$0008	zero = The CIC protocol is disabled. non-zero = The CIC protocol is enabled. Refer to the NI-488.2 user manual for more information about device-level calls and bus management.
IbaIRQ	$0009	zero = Interrupts are not enabled. non-zero = Interrupts are enabled.
IbaSC	$000A	zero = The board is not the GPIB System Controller. non-zero = The board is the System Controller. See `ibrsc`.
IbaSRE	$000B	zero = The board does not automatically assert the GPIB REN line when it becomes the System Controller. non-zero = The board automatically asserts REN when it becomes the System Controller. See `ibrsc` and `ibsre`.

(continues)

IBASK (continued) IBASK

Table 3-1. ibask Board Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Returned Information
IbaEOSrd	$000C	zero = The EOS character is ignored during read operations. non-zero = Read operation is terminated by the EOS character. See `ibeos`.
IbaEOSwrt	$000D	zero = The EOI line is not asserted when the EOS character is sent during a write operation. non-zero = The EOI line is asserted when the EOS character is sent during a write operation. See `ibeos`.
IbaEOScmp	$000E	zero = A 7-bit compare is used for all EOS comparisons. non-zero = An 8-bit compare is be used for all EOS comparisons. See `ibeos`.
IbaEOSchar	$000F	The current EOS character of the board. See `ibeos`.
IbaPP2	$0010	zero = The board is in PP1 mode–remote parallel poll configuration. non-zero = The board is in PP2 mode–local parallel poll configuration. Refer to the NI-488.2 user manual for more information about parallel polls.

(continues)

IBASK (continued) IBASK

Table 3-1. ibask Board Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Returned Information
IbaTIMING	$0011	The current bus timing of the board. 1 = Normal timing (T1 delay of 2 µs.) 2 = High speed timing (T1 delay of 500 ns.) 3 = Very high speed timing (T1 delay of 350 ns.)
IbaDMA	$0012	zero = The board will not use DMA for GPIB transfers. non-zero = The board will use DMA for GPIB transfers. See `ibdma`.
IbaReadAdjust	$0013	0 = Read operations do not have pairs of bytes swapped. 1 = Read operations have each pair of bytes swapped.
IbaWriteAdjust	$0014	0 = Write operations do not have pairs of bytes swapped. 1 = Write operations have each pair of bytes swapped.
IbaEventQueue	$0015	zero = The event queue is disabled. non-zero = The event queue is enabled. See `ibevent`.

(continues)

IBASK (continued) IBASK

Table 3-1. ibask Board Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Returned Information
IbaSpollBit	$0016	zero = The SPOLL bit of `ibsta` is disabled. non-zero = The SPOLL bit of `ibsta` is enabled. See the NI-488.2 user manual for information about Talker/Listener applications.
IbaSendLLO	$0017	zero = The GPIB LLO command is not sent when a device is put online-`ibfind` or `ibdev`. non-zero = The LLO command is sent.
IbaPPollTime	$0019	0 = The board uses the standard duration (2 µs) when conducting a parallel poll. 1 to 17 = The board uses a variable length duration when conducting a parallel poll. The duration values correspond to the `ibtmo` timing values.
IbaEndBitIsNormal	$001A	zero = The END bit of `ibsta` is set only when EOI or EOI plus the EOS character is received. If the EOS character is received without EOI, the END bit is not set. non-zero = The END bit is set whenever EOI, EOS, or EOI plus EOS is received.

(continues)

IBASK (continued) IBASK

Table 3-1. ibask Board Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Returned Information
IbaHSCableLength	$001F	0 = High-speed data transfer (HS488) is disabled. 1 to 15 = High-speed data transfer (HS488) is enabled. The number returned represents the number of meters of GPIB cable in your system. See the NI-488.2 user manual for information about high-speed data transfers (HS488).
IbaBaseAddr	$0201	The base I/O address of the board.
IbaDmaChannel	$0202	The DMA channel that the board is configured to use. If the board is not configured to use DMA, the error ECAP is returned.
IbaIrqLevel	$0203	The interrupt level that the board is configured to use. If the board is not configured to use interrupts, the error ECAP is returned.
IbaBaud	$0204	If your GPIB interface is GPIB-232CT-A, then this option returns the baud rate that the NI-488.2 software is configured to use when communicating with the interface.
IbaParity	$0205	If your GPIB interface is GPIB-232CT-A, then this option returns the parity that the NI-488.2 software is configured to use when communicating with the interface.

(continues)

IBASK (continued) IBASK

Table 3-1. ibask Board Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Returned Information
IbaStopBits	$0206	If your GPIB interface is GPIB-232CT-A, then this option returns the number of stop bits that the NI-488.2 software is configured to use when communicating with the interface.
IbaDataBits	$0207	If your GPIB interface is GPIB-232CT-A, then this option returns the number of data bits that the NI-488.2 software is configured to use when communicating with the interface.
IbaComPort	$0208	If your GPIB interface is GPIB-232CT-A, then this option returns the COM port number that the NI-488.2 software is configured to use when communicating with the interface.
IbaComIrqLevel	$0209	If your GPIB interface is GPIB-232CT-A, then this option returns the interrupt level that the NI-488.2 software is configured to use when communicating with your computer serial port. If the software is not directly accessing the serial port, the error ECAP is returned.

(continues)

IBASK (continued) IBASK

Table 3-1. ibask Board Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Returned Information
IbaComPortBase	$020A	If your GPIB interface is GPIB-232CT-A, then this option returns the base I/O address that the NI-488.2 software is configured to use when communicating with your computer serial port. If the software is not directly accessing the serial port, the error ECAP is returned.
IbaSingleCycleDma	$020B	If your GPIB interface is an AT-GPIB, then this option returns the DMA transfer mode that the NI-488.2 software is configured to use when performing DMA transfers. zero = demand mode DMA non-zero = single cycle DMA See your getting started manual for more information.
IbaSocketNumber	$020C	If your GPIB interface is PCMCIA-GPIB, this option returns the number of the socket where the interface is inserted.

IBASK (continued) IBASK

Table 3-2. ibask Device Configuration Parameter Options

Options (Constants)	Options (Values)	Returned Information
IbaPAD	$0001	The current primary address of the device. See `ibpad`.
IbaSAD	$0002	The current secondary address of the device. See `ibsad`.
IbaTMO	$0003	The current I/O timeout of the device. See `ibtmo`.
IbaEOT	$0004	zero = The GPIB EOI line is not asserted at the end of a write operation. non-zero = EOI is asserted at the end of a write operation. See `ibeot`.
IbaREADDR	$0006	zero = No unnecessary addressing is performed between device-level read and write operations. non-zero = Addressing is always performed before a device-level read or write operation. See `ibeot`.
IbaEOSrd	$000C	zero = The EOS character is ignored during read operations. non-zero = Read operation is terminated by the EOS character. See `ibeos`.

(continues)

IBASK (continued) IBASK

Table 3-2. ibask Device Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Returned Information
IbaEOSwrt	$000D	zero = The EOI line is not asserted when the EOS character is sent during a write operation. non-zero = The EOI line is asserted when the EOS character is sent during a write operation. See `ibeos`.
IbaEOScmp	$000E	zero = A 7-bit compare is used for all EOS comparisons. non-zero = An 8-bit compare is used for all EOS comparisons. See `ibeos`.
IbaEOSchar	$000F	The current EOS character of the device. See `ibeos`.
IbaReadAdjust	$0013	0 = Read operations do not have pairs of bytes swapped. 1 = Read operations have each pair of bytes swapped.
IbaWriteAdjust	$0014	0 = Write operations do not have pairs of bytes swapped. 1 = Write operations have each pair of bytes swapped.

(continues)

IBASK (continued) IBASK

Table 3-2. ibask Device Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Returned Information
IbaSPollTime	$0018	The length of time the driver waits for a serial poll response when polling the device. The length of time is represented by the `ibtmo` timing values.
IbaEndBitIsNormal	$001A	zero = The END bit of `ibsta` is set only when EOI or EOI plus the EOS character is received. If the EOS character is received without EOI, the END bit is not set. non-zero = The END bit is set whenever EOI, EOS, or EOI plus EOS is received.
IbaUnAddr	$001B	zero = The GPIB commands Untalk (UNT) and Unlisten (UNL) are not sent after each device-level read and write operation. non-zero = The UNT and UNL commands are sent after each device-level read and write operation.
IbaBNA	$0200	The index of the GPIB access board used by the given device descriptor.

IBBNA IBBNA

Purpose: Change access board of device.

Format: `ibbna (ud,bname)`

`ud` represents a device. `bname` is the new access board to be used in all device calls to that device. For MS-DOS Pascal `bname` is of type `nbuf` as defined in the declaration files. For Turbo Pascal for Windows, `bname` is of type `PChar`.

`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.

Device Function Example:

Associate the device `dvm` with the interface board `GPIB0`.

MS-DOS Pascal

```
var devname,bdname : nbuf;
               dvm : integer;
devname := 'DVM    ';
dvm := ibfind (devname);
(* Set GPIB0 as the access board for device dvm.    *)
bdname := 'GPIB0  ';
ibbna (dvm,bdname);
```

Turbo Pascal for Windows

```
var devname,bdname : PChar;
               dvm : integer;
devname := 'DVM';
dvm := ibfind (devname);
(* Set GPIB0 as the access board for device dvm.    *)
bdname := 'GPIB0';
ibbna (dvm,bdname);
```

IBCAC IBCAC

Purpose: Become Active Controller.

Format: `ibcac (ud,v)`

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

To take control synchronously, the GPIB board asserts the ATN signal without the transfer of corrupting data. 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 impossible to gain control synchronously (for example, after a timeout error).

It is generally not necessary to use the `ibcac` function in most applications. The GPIB board automatically takes control in functions such as `ibcmd` and `ibrpp`.

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.

```
var brd0 : integer;
ibcac (brd0,0);          (* ibsta should show        *)
                         (* that the interface       *)
                         (* board is now CAC         *)
                         (* (CIC with ATN)           *)
```

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

MS-DOS Pascal

```
var     rd : cbuf;
    bdname : nbuf;
    brd0   : integer;
bdname := 'GPIB0  ';
brd0   := ibfind (bdname);
ibrd (brd0,rd,255);
ibcac (brd0,1);
```

Turbo Pascal for Windows

```
var     rd : array[0..256] of char;
    bdname : PChar;
    brd0   : integer;
bdname := 'GPIB0';
brd0   := ibfind (bdname);
ibrd (brd0,rd,255);
ibcac (brd0,1);
```

IBCLR IBCLR

Purpose: Clear specified device.

Format: `ibclr (ud)`

`ud` represents a device.

The `ibclr` function clears the internal or device functions of a specified device. `ibclr` calls the board function `ibcmd` to send each of 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.

Device Function Example:

Clear the device `vmtr`.

MS-DOS Pascal

```
var devname : nbuf;
       vmtr : integer;
devname := 'DEV3   ';        (* open voltmeter     *)
vmtr := ibfind (devname);
ibclr (vmtr);                (* clear voltmeter    *)
```

Turbo Pascal for Windows

```
var devname : PChar;
       vmtr : integer;
devname := 'DEV3';           (* open voltmeter     *)
vmtr := ibfind (devname);
ibclr (vmtr);                (* clear voltmeter    *)
```

IBCMD IBCMD

Purpose: Send GPIB command messages.

Format: `ibcmd (ud,cmd,cnt)`

`ud` represents an interface board. The pre-allocated array `cmd` contains the commands to be sent over the GPIB. `cnt` is the number of commands sent. The `cnt` value is of type `integer4` in Microsoft Pascal, and of type `longINT` in QuickPascal, Turbo Pascal, and Turbo Pascal for Windows. The `cnt` value is of type `integer` in IBM Pascal.

The `ibcmd` function is used to transmit interface messages (commands) over the GPIB. These commands are listed in Appendix A. The `ibcmd` function is also used to pass GPIB control to another device. This function is *not* used to transmit programming instructions to devices. Programming 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 terminating events listed above except the first.

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

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 corresponding to a

IBCMD (continued) IBCMD

numeric value. Refer to Appendix A for ASCII characters and corresponding numeric values.

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).

   Pascal

```
var cmd : cbuf;
cmd[1] := chr(UNL);    (* GPIB Unlisten command     *)
cmd[2] := 'F';         (* talk address              *)
cmd[3] := '1';         (* listen address            *)
ibcmd (brd0,cmd,3);
```

   Turbo Pascal for Windows

```
var cmd : PChar;
cmd := '?F1';
ibcmd (brd0,cmd^,3);
```

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

   Pascal

```
var cmd : cbuf;
cmd[1] := chr(UNL);    (* GPIB Unlisten command     *)
cmd[2] := 'F';         (* talk address              *)
cmd[3] := '1';         (* listen address            *)
cmd[4] := 'n';         (* secondary address         *)
ibcmd (brd0,cmd,4);
```

   Turbo Pascal for Windows

```
var cmd : PChar;
cmd := '?F1n';
ibcmd (brd0,cmd^,4);
```

IBCMD (continued) IBCMD

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

   Pascal

```
var cmd : cbuf;
cmd[1] := chr(DCL);
ibcmd (brd0,cmd,1);
```

   Turbo Pascal for Windows

```
var cmd : PChar;
cmd := #14;
ibcmd (brd0,cmd^,1);
```

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.

   Pascal

```
var cmd : cbuf;
cmd[1] := chr(UNL);    (* GPIB Unlisten command     *)
cmd[2] := '!';         (* First listen address      *)
cmd[3] := '(';         (* Second listen address     *)
cmd[4] := chr(SDC);    (* Selected Device Clear     *)
ibcmd (brd0,cmd,4);
```

   Turbo Pascal for Windows

```
var cmd : PChar;
cmd := '?!('#04;
ibcmd (brd0,cmd^,4);
```

IBCMD (continued) IBCMD

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

Pascal

```
var cmd : cbuf;
cmd[1] := chr(GET);
ibcmd (brd0,cmd,1);
```

Turbo Pascal for Windows

```
var cmd : PChar;
cmd := #08;
ibcmd (brd0,cmd^,1);
```

6. Unaddress all Listeners and 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).

Pascal

```
var rd, cmd : cbuf;
cmd[1] := chr(UNL);    (* GPIB Unlisten command     *)
cmd[2] := 'R';         (* talk address              *)
cmd[3] := ' ';         (* listen address            *)
cmd[4] := chr(SPE);    (* Serial Poll Enable        *)
ibcmd (brd0,cmd,4);
ibrd (brd0,rd,1);
(* 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.                           *)
cmd[1] := chr(SPD);    (* Serial Poll Disable       *)
cmd[2] := chr(UNT);    (* GPIB Untalk command       *)
ibcmd (brd0,cmd,2);
```

IBCMD (continued) IBCMD

Turbo Pascal for Windows

```
var rd : array[1..10] of char;
   cmd : PChar;
cmd := '?R '#18;
ibcmd (brd0,cmd^,4);
ibrd (brd0,rd,1);
(* 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.                           *)
cmd := #19'_';
ibcmd (brd0,cmd^,2);
```

IBCMDA IBCMDA

Purpose: Send commands asynchronously from string.

Format: `ibcmda (ud,cmd,cnt)`

`ud` represents an interface board. The pre-allocated array `cmd` contains the commands to be sent over the GPIB. `cnt` is the number of commands sent. The `cnt` value is of type `integer4` in Microsoft Pascal, and of type `longINT` in QuickPascal, Turbo Pascal, and Turbo Pascal for Windows. The `cnt` value is of type `integer` in IBM Pascal.

The `ibcmda` function is used to transmit interface messages (commands) over the GPIB. These commands are listed in Appendix A. 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.

Use `ibcmda` instead 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 an 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.

IBCMDA (continued) IBCMDA

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

Note: Resynchronization is successful only 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.

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

IBCMDA (continued) IBCMDA

Board Function Example:

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

IBM/MS Pascal

```
var brd0 : integer;
  bdname : nbuf;
     cmd : cbuf;
    mask : word;
(* 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.                  *)
bdname := 'GPIB0  ';
brd0 := ibfind (bdname);
cmd[1]  := '?';        (* GPIB Unlisten command    *)
cmd[2]  := '@';        (* my talk address          *)
cmd[3]  := '1';        (* listen address 1         *)
cmd[4]  := '2';        (* listen address 2         *)
cmd[5]  := '3';        (* listen address 3         *)
cmd[6]  := '4';        (* listen address 4         *)
cmd[7]  := '5';        (* listen address 5         *)
cmd[8]  := '6';        (* listen address 6         *)
cmd[9]  := '7';        (* listen address 7         *)
cmd[10] := '8';        (* listen address 8         *)
cmd[11] := '9';        (* listen address 9         *)
ibcmda (brd0,cmd,11);
While ibsta and 16#100 <> 16#100 do
Begin
     eventtst;          (* Unspecified routine     *)
                        (* to test and process     *)
                        (* a high priority         *)
     ibwait (brd0,0);   (* event.  Set mask to     *)
                        (* return immediately.     *)
     IF ibsta < 0 then error;
End;

writeln ('Asynchronous commands sent!');
mask :=16#4100;
ibwait (brd0, mask);
write ('Asynchronous transfer properly terminated');
```

IBCMDA (continued) IBCMDA

QuickPascal/Turbo Pascal

```
var brd0 : integer;
  bdname : nbuf;
     cmd : cbuf;
    mask : word;
(* $40, addresses nine Listeners at addresses      *)
(* $31 through $39 to receive the broadcast        *)
(* message.                                        *)
bdname := 'GPIB0  ';
brd0 := ibfind (bdname);
cmd[1]  := '?';        (* GPIB Unlisten command    *)
cmd[2]  := '@';        (* my talk address          *)
cmd[3]  := '1';        (* listen address 1         *)
cmd[4]  := '2';        (* listen address 2         *)
cmd[5]  := '3';        (* listen address 3         *)
cmd[6]  := '4';        (* listen address 4         *)
cmd[7]  := '5';        (* listen address 5         *)
cmd[8]  := '6';        (* listen address 6         *)
cmd[9]  := '7';        (* listen address 7         *)
cmd[10] := '8';        (* listen address 8         *)
cmd[11] := '9';        (* listen address 9         *)
ibcmda (brd0,cmd,11);
While ibsta and $100 <> $100 do
Begin
     eventtst;          (* Unspecified routine     *)
                        (* to test and process     *)
                        (* a high priority         *)
     ibwait (brd0,0);   (* event.  Set mask to     *)
                        (* return immediately.     *)
     IF ibsta < 0 then error;
End;

writeln ('Asynchronous commands sent!');
mask := $4100;
ibwait (brd0, mask);
write ('Asynchronous transfer properly terminated');
```

IBCMDA (continued) IBCMDA

Turbo Pascal for Windows

```
var brd0     : integer;
  bdname,cmd : PChar;
    mask : word;
(* The interface board brd0, at talk address       *)
(* $40 (ASCII @), addresses nine Listeners         *)
(* at addresses $31-$39 (ASCII 1-9) to             *)
(* receive the broadcast message.                  *)
bdname := 'GPIB0';
brd0 := ibfind (bdname);
(* GPIB Unlisten command my talk address           *)
(* listen address 1 through listen address 9       *)
cmd := '?@123456789';
ibcmda (brd0,cmd^,11);
While (ibsta and $100) <> $100 do
Begin
     eventtst;          (* Unspecified routine     *)
                        (* to test and process     *)
                        (* a high priority         *)
     ibwait (brd0,0);   (* event.  Set mask to     *)
                        (* return immediately.     *)
     IF ibsta < 0 then error;
End;

writeln ('Asynchronous commands sent!');
mask := $4100;
ibwait (brd0, mask);
writeln ('Asynchronous transfer properly terminated');
```

IBCONFIG IBCONFIG

Purpose: Change the driver configuration parameters.

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

`ud` represents 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 are returned in `iberr`. Table 3-3 shows the values of `option` and `value` that are available when `ud` is a GPIB interface board descriptor. Table 3-4 shows the values that are available when `ud` is a device descriptor.

Table 3-3. ibconfig Board Configuration Parameter Options

Options (Constants)	Options (Values)	Legal Values
IbcPAD	$0001	Changes the primary address of the board. Identical to `ibpad`. Default determined by `ibconf`.
IbcSAD	$0002	Changes the secondary address of the board. Identical to `ibsad`. Default determined by `ibconf`.
IbcTMO	$0003	Changes the I/O timeout limit of the board. Identical to `ibtmo`. Default determined by `ibconf`.
IbcEOT	$0004	Changes the data termination mode for write operations. Identical to `ibeot`. Default determined by `ibconf`.
IbcPPC	$0005	Configures the board for parallel polls. Identical to board-level `ibppc`. Default: zero.

(continues)

IBCONFIG (continued) IBCONFIG

Table 3-3. ibconfig Board Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Legal Values
IbcAUTOPOLL	$0007	zero = Disable automatic serial polling. non-zero = Enable automatic serial polling. Default determined by `ibconf`. Refer to the NI-488.2 user manual for more information about automatic serial polling.
IbcCICPROT	$0008	zero = Disable the CIC protocol. non-zero = Enable the CIC protocol. Default determined by `ibconf`. Refer to the NI-488.2 user manual for more information about the CIC protocol.
IbcIRQ	$0009	zero = Do not use interrupts. non-zero = Use interrupts-use the hardware interrupt level configured through `ibconf`. Default determined by `ibconf`.
IbcSC	$000A	Request or release system control. Identical to `ibrsc`. Default determined by `ibconf`.
IbcSRE	$000B	Assert the Remote Enable (REN) line. Identical to `ibsre`. Default: zero.
IbcEOSrd	$000C	zero = Ignore EOS character during read operations. non-zero = Terminate reads when the EOS character is read match occurs. Default determined by `ibconf`.

(continues)

IBCONFIG (continued) IBCONFIG

Table 3-3. ibconfig Board Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Legal Values
IbcEOSwrt	$000D	zero = Do not assert EOI with the EOS character during write operations. non-zero = Assert EOI with the EOS character during writes operations. Default determined by `ibconf`.
IbcEOScmp	$000E	zero = Use 7 bits for the EOS character comparison. non-zero = Use 8 bits for the EOS character comparison. Default determined by `ibconf`.
IbcEOSchar	$000F	Any 8-bit value. This byte becomes the new EOS character. Default determined by `ibconf`.
IbcPP2	$0010	zero = PP1 mode-remote parallel poll configuration. non-zero = PP2 mode-local parallel poll configuration. Default: zero. Refer to the NI-488.2 user manual for more information about parallel polling.
IbcTIMING	$0011	1 = Normal timing (T1 delay of 2 µs). 2 = High-speed timing (T1 delay of 500 ns). 3 = Very high-speed timing (T1 delay of 350 ns). Default determined by `ibconf`. The T1 delay is the GPIB source handshake timing.

(continues)

IBCONFIG (continued) IBCONFIG

Table 3-3. ibconfig Board Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Legal Values
IbcDMA	$0012	Identical to `ibdma`. Default determined by `ibconf`.
IbcReadAdjust	$0013	0 = No byte swapping. 1 = Swap pairs of bytes during a read. Default: zero.
IbcWriteAdjust	$0014	0 = No byte swapping. 1 = Swap pairs of bytes during a write. Default: zero.
IbcEventQueue	$0015	zero = The event queue is disabled. non-zero = The event queue is enabled. Default: zero. See `ibevent`.
IbcSpollBit	$0016	zero = The SPOLL bit of `ibsta` is disabled. non-zero = The SPOLL bit of `ibsta` is enabled. Default: zero. See NI-488.2 user manual for information about Talker/Listener applications.
IbcSendLLO	$0017	zero = Do not send LLO when putting a device online –`ibfind` or `ibdev`. non-zero = Send LLO when putting a device online–`ibfind` or `ibdev`. Default: zero.

(continues)

IBCONFIG (continued) IBCONFIG

Table 3-3. ibconfig Board Configuration Parameter Options (continued)

Options (Constants)	Options (Values)	Legal Values
IbcPPollTime	$0019	0 = Use the standard duration (2 µs) when conducting a parallel poll. 1 to 17 = Use a variable length duration when conducting a parallel poll. The duration represented by 1 to 17 corresponds to the `ibtmo` values. Default: zero.
IbcEndBitIsNormal	$001A	zero = Do not set the END bit of `ibsta` when an EOS match occurs during a read. non-zero = Set the END bit of `ibsta` when an EOS match occurs during a read. Default: non-zero.
IbcHSCableLength	$001F	0 = High-speed data transfer (HS488) is disabled. 1 to 15 = The number of meters of GPIB cable in your system. The NI-488.2 software uses this information to select the appropriate high-speed data transfer (HS488) mode. Default determined by `ibconf`. See the NI-488.2 user manual for information about high-speed data transfers (HS488).

IBCONFIG (continued) IBCONFIG

Table 3-4. ibconfig Device Configuration Parameter Options

Options (Constants)	Options (Values)	Legal Values
IbcPAD	$0001	Changes the primary address of the device. Identical to `ibpad`. Default determined by `ibconf`.
IbcSAD	$0002	Changes the secondary address of the device. Identical to `ibsad`. Default determined by `ibconf`.
IbcTMO	$0003	Changes the device I/O timeout limit. Identical to `ibtmo`. Default determined by `ibconf`.
IbcEOT	$0004	Changes the data termination method for writes. Identical to `ibeot`. Default determined by `ibconf`.
IbcREADDR	$0006	zero = No unnecessary readdressing is performed between device-level reads and writes. non-zero = Addressing is always performed before a device-level read or write. Default determined by `ibconf`.
IbcEOSrd	$000C	non-zero = Terminate reads when the EOS character is read. Default determined by `ibconf`.
IbcEOSwrt	$000D	zero = Do not send EOI with the EOS character during write operations. non-zero = Send EOI with the EOS character during writes. Default determined by `ibconf`.

(continues)

IBCONFIG (continued) IBCONFIG

Table 3-4. ibconfig Device Configuration Parameter Options (continued)

IbcEOScmp	$000E	zero = Use seven bits for the EOS character comparison. non-zero = Use 8 bits for the EOS character comparison. Default determined by `ibconf`.
IbcEOSchar	$000F	Any 8-bit value. This byte becomes the new EOS character. Default determined by `ibconf`.
IbcReadAdjust	$0013	0 = No byte swapping. 1 = Swap pairs of bytes during a read. Default: zero.
IbcWriteAdjust	$0014	0 = No byte swapping. 1 = Swap pairs of bytes during a write. Default: zero.
IbcSPollTime	$0018	0 to 17 = Sets the length of time the driver waits for a serial poll response byte when polling the given device. The length of time represented by 0 to 17 corresponds to the `ibtmo` values. Default: 11.
IbcEndBitIsNormal	$001A	zero = Do not set END bit of `ibsta` when EOS match occurs during a read. non-zero = Set END bit of `ibsta` when EOS match occurs during a read. Default: non-zero.
IbcUnAddr	$001B	zero = Do not send Untalk (UNT)and Unlisten (UNL)–at the end of device-level reads and writes. non-zero = Send UNT and UNL at the end of device-level reads and writes. Default: zero.

Device Function Example:

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

MS-DOS Pascal

```
var   dvm : integer;
  devname : nbuf;
devname := 'Dev1   ';
dvm := ibfind (devname);
(* Enable repeat addressing.                          *)
ibconfig (dvm,6,1);
(* Set linefeed as the EOS character.                 *)
ibconfig (dvm,15,10);
(* Use 7-bit comparison for EOS character.            *)
ibconfig (dvm,14,0);
(* Terminate reads on EOS.                            *)
ibconfig (dvm,12,1);
```

Turbo Pascal for Windows

```
var   dvm : integer;
  devname : PChar;
devname := 'Dev1';
dvm := ibfind (devname);
(* Enable repeat addressing.                          *)
ibconfig (dvm,6,1);
(* Set linefeed as the EOS character.                 *)
ibconfig (dvm,15,10);
(* Use 7-bit comparison for EOS character.            *)
ibconfig (dvm,14,0);
(* Terminate reads on EOS.                            *)
ibconfig (dvm,12,1);
```

IBCONFIG (continued) IBCONFIG

Board Function Examples:

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

   MS-DOS Pascal

```
var   brd0 : integer;
    bdname : nbuf;
bdname := 'gpib0  ';
brd0 := ibfind (bdname);
(* Enable DMA transfers.                              *)
ibconfig (brd0,IbcDMA,1);
(* Turn off autopolling.                              *)
ibconfig (brd0,IbcAUTOPOLL,0);
(* Turn on interrupts.                                *)
ibconfig (brd0,IbcIRQ,1);
```

   Turbo Pascal for Windows

```
var   brd0 : integer;
brd0 := ibfind ('gpib0');
(* Enable DMA transfers.                              *)
ibconfig (brd0,IbcDMA,1);
(* Turn off autopolling.                              *)
ibconfig (brd0,IbcAUTOPOLL,0);
(* Turn on interrupts.                                *)
ibconfig (brd0,IbcIRQ,1);
```

2. Enable automatic byte swapping of binary integer data.

   IBM/MS Pascal/QuickPascal

```
var array : ibuf;
   header : cbuf;
(* Read in unswapped header data.                     *)
ibrd (ud,header,10);
(* Arrange for byte swapping.                         *)
ibconfig (ud,IbcReadAdjust,1);
(* Read 1,000 bytes with automatic swapping.          *)
ibrdi (ud,array,1000);
(* Disable swapping for subsequent reads.             *)
ibconfig (ud,IbcReadAdjust,0);
```

IBCONFIG (continued) IBCONFIG

Turbo Pascal/Turbo Pascal for Windows

```
var array : array[1..500] of integer;
   header : array[1..10] of char;
(* Read in unswapped header data.                    *)
ibrd (ud,header,10);
(* Arrange for byte swapping.                        *)
ibconfig (ud,IbcReadAdjust,1);
(* Read 1,000 bytes with automatic swapping.         *)
ibrd (ud,array,1000);
(* Disable swapping for subsequent reads.            *)
ibconfig (ud,IbcReadAdjust,0);
```

IBDEV IBDEV

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

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

`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 data 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 unopened 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-configurable 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 of the `ibonl` function 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

Board/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` = `16#67` or `$67`), a timeout of 10 msec (`tmo` = 7), the END message enabled (`eot` = 1) and the EOS mode disabled (`eos` = 0).

IBM/MS Pascal

```
(* Get a device descriptor associated with          *)
(* board 0.                                         *)
ud := ibdev (0,6,16#67,T10s,1,0);
if ud < 0 then
   begin
   (* Handle GPIB error here.                       *)
   if iberr = EDVR then
      begin
      (* Bd is not correct or no devices are       *)
      (* available.                                *)
      end
   else
      if iberr = EARG then
      begin
      (* The call succeeded, but at least          *)
      (* one of pad, sad, tmo, eos, or eot is      *)
      (* incorrect.                                *)
      end;
   end;
```

IBDEV (continued) IBDEV

QuickPascal/Turbo Pascal/Turbo Pascal for Windows

```
(* Get a device descriptor associated with          *)
(* board 0.                                         *)
ud := ibdev (0,6,$67,T10s,1,0);
if ud < 0 then
    begin
    (* Handle GPIB error here.                      *)
    if iberr = EDVR then
        begin
        (* Bd is not correct or no devices are      *)
        (* available.                               *)
        end
    else
        if iberr = EARG then
        begin
        (* The call succeeded, but at least         *)
        (* one of pad, sad, tmo, eos, or eot is     *)
        (* incorrect.                               *)
        end;
    end;
```

IBDMA IBDMA

Purpose: Enable or disable DMA.

Format: `ibdma (ud,v)`

`ud` represents 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 (0), programmed I/O is used.

If you enabled DMA at configuration time, this function can be used to switch between programmed I/O and the selected DMA 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 either `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`.

Board Function Examples:

1. Enable DMA transfers using the previously configured channel.

```
var v : integer;
v := 1;    (* Any non-zero value will do.          *)
ibdma (brd0,v);
```

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

```
var v : integer;
v := 0;
ibdma (brd0,v);
```

IBEOS IBEOS

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

Format: `ibeos (ud,v)`

`ud` represents a device or an interface board. `v` specifies the EOS character and the data transfer termination method according to Table 3-5. `ibeos` is needed only to alter the value from its configuration setting.

The assignment made by this function remains in effect until either `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 3-5 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.

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.

IBEOS (continued) IBEOS

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.

Device Function Example:

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

```
var dvm,v : integer;
      wrt : cbuf;
v := XEOS + LF;          (* EOS information        *)
                         (* for IBEOS              *)
ibeos (dvm,v);
wrt [1] := '1';          (* Data bytes to be       *)
wrt [2] := '2';          (* written                *)
wrt [3] := '3';
wrt [4] := chr (LF);     (* The EOS character      *)
                         (* is the last byte.      *)
ibwrt (dvm,wrt,4);
```

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.

```
var  v : integer;
    rd : cbuf;
v := REOS + LF;              (* The END bit in      *)
ibeos (brd0,v);              (* ibsta is set if     *)
ibrd (brd0,rd,200);          (* the read            *)
                             (* terminated on       *)
                             (* the EOS             *)
                             (* character.          *)
                             (* Assume board has    *)
                             (* been addressed      *)
                             (* to do board         *)
                             (* read.               *)
```

2. Program the interface board `brd0` to terminate read operations on the 8-bit value hex 82 rather than the 7-bit character hex 0A. Change `v` used in Example 1.

IBM/MS Pascal

```
var  v : integer;
    rd : cbuf;
v := BIN + REOS + 16#82;
ibeos (brd0,v);
ibrd (brd0,rd,200);
```

QuickPascal/Turbo Pascal/Turbo Pascal for Windows

```
var  v : integer;
    rd : cbuf;
v := BIN + REOS + $82;
ibeos (brd0,v);
ibrd (brd0,rd,200);
```

IBEOS (continued) IBEOS

3. Disable read termination on receiving the EOS character for operations involving the interface board `brd0`. Change `v` used in Example 1.

```
var  v : integer;
    rd : cbuf;
v := 0;                 (* No EOS modes enabled.    *)
ibeos (brd0,v);
ibrd (brd0,rd,200);
```

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

```
var  v : integer;
   wrt : cbuf;
v := XEOS + LF;       (* EOS information for        *)
                      (* IBEOS.                     *)
ibeos (brd,v);
wrt[1] := '1';        (* Data bytes to be           *)
wrt[2] := '2';        (* written.                   *)
wrt[3] := '3';
wrt[4] := chr(LF);    (* EOS character is the       *)
                      (* last byte.                 *)
ibwrt (brd0,wrt,4);
```

5. Send END with linefeeds and terminate reads on linefeeds for operations involving the interface board `brd0`. Change `v` used in Example 4.

```
var  v : integer;
   wrt : cbuf;
v := REOS + XEOS + LF;    (* EOS information        *)
                          (* for IBEOS.             *)
ibeos (brd0,v);
wrt[1] := '1';            (* Data bytes to be       *)
wrt[2] := '2';            (* written.               *)
wrt[3] := '3';
wrt[4] := chr(LF);        (* EOS character is       *)
                          (* the last byte.         *)
ibwrt (brd0,wrt,4);
```

IBEOT IBEOT

Purpose: Enable/disable END message on write operations.

Format: `ibeot (ud,v)`

`ud` represents 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 (0), END is not automatically sent.

The END message is the assertion of the GPIB EOI signal. If the automatic END termination message is enabled (`v` is non-zero), it is not necessary to use the EOS character to identify the last byte of a data string. Sending END with the EOS character is controlled by the `ibeos` function and is not affected by `ibeot`.

`ibeot` is used to send variable length data or to alter the value from the default configuration setting. (In the default configuration, this feature is enabled.) The assignment made by this function remains in effect until either `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 (1) if automatic END message was previously enabled, or with a zero (0) 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 of the device.

IBEOT (continued) IBEOT

Device Function Example:

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

```
var  wrt : cbuf;
ibeot (plotter,1);
                    (* It is assumed that wrt      *)
                    (* contains the data to be     *)
                    (* written to the GPIB.        *)
ibwrt (plotter,wrt,3);
```

Board Function Examples:

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

   ```
   ibeot (brd0,0);
   ```

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

   ```
   var  wrt : cbuf;
   ibeot (brd0,1);
                         (* It is assumed that       *)
                         (* wrt contains the data    *)
                         (* to be written and        *)
                         (* that all Listeners       *)
   ibwrt (brd0,wrt,3);   (* have been addressed.     *)
   ```

IBEVENT IBEVENT

Purpose: Return the next event.

Format: `ibevent (ud,event)`

`ud` specifies an interface board. `event` stores the event code.

The `ibevent` function is used to determine which GPIB event (Device Clear or Device Trigger) occurred. It is normally called when the EVENT bit has been set in `ibsta`. The variable `event` is filled in with one of the following values:

0 = No events are in the event queue.

1 = A Device Clear message was received.

2 = A Device Trigger message was received.

Upon returning from this function, `ibcnt` contains the number of events that remain in the event queue.

This function is typically used in talker/listener (T/L) applications, not controller applications. Often a T/L application must determine the order in which Device Clear and Device Trigger messages are received by the interface board. The usual DCAS and DTAS bits of `ibsta` are not sufficient in determining the order of the events. When the event queue is enabled (by using the `ibconfig` function to enable the EVENT bit of `ibsta`), any time the driver receives a DCAS or DTAS message, the event is stored in the event queue of the board and the EVENT bit is set in `ibsta`. If any I/O is in progress, it is stopped with the error EABO. The application program can then call `ibevent` to determine which event or events occurred and process those events. The event queue must be empty (the EVENT must not be in `ibsta`) before any more I/O can be started.

If the event queue fills up, a call to `ibevent` returns the ETAB error along with the oldest event in the queue.

IBEVENT (continued) IBEVENT

Board Function Example:

Determine which GPIB event occurred

```
var mask : word;
   event : integer;
mask := TIMO or EVENT;       (* mnemonics defined  *)
                             (* in header files    *)
ibwait (brd0,mask);
IF(ibsta and EVENT) <> 0 then ibevent (brd0,event);
(* event contains the event code                   *)
```

IBFIND IBFIND

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

Format: `ud := ibfind (udname)`

`ud` is a variable containing the unit descriptor returned by `ibfind`. The number referred to throughout this manual as a unit descriptor is returned here in the variable `ud`. `udname` is a string containing a default or configured device or board name. The name used in the `udname` argument must match the default or configured device or board name. `udname` is of type `nbuf` as defined in the MS-DOS Pascal header files. For Turbo Pascal for Windows `udname` is of type `PChar`.

`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. Use a variable name close to the actual name of the device or board to simplify programming effort.

Note: For board calls, the unit descriptor may be substituted with an integer board index 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.

`ibfind` performs the equivalent of `ibonl` to open the specified device or board and to initialize software parameters to their default configuration settings. The unit descriptor returned in `ud` 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.

MS-DOS Pascal

```
var     dvm : integer;
    devname : nbuf;
devname := 'DEV4   ';        (* Device name          *)
                             (* assigned at          *)
                             (* configuration        *)
                             (* time.                *)
dvm := ibfind (devname);     (* If dvm < 0, an       *)
if dvm < 0 then error;       (* error occurred.      *)
```

Turbo Pascal for Windows

```
var     dvm : integer;
    devname : PChar;
devname := 'DEV4';           (* Device name          *)
                             (* assigned at          *)
                             (* configuration        *)
                             (* time.                *)
dvm := ibfind (devname);     (* If dvm < 0, an       *)
if dvm < 0 then error;       (* error occurred.      *)
```

IBFIND (continued) IBFIND

Board Function Example:

Assign the unit descriptor associated with the interface board `GPIB0` to the variable `brd0`.

MS-DOS Pascal

```
var  brd0 : integer;
   bdname : nbuf;
bdname := 'GPIB0  ';          (* Factory default     *)
                              (* board name.         *)
brd0 := ibfind (bdname);
if brd0 < 0 then error;       (* If brd0 < 0, an     *)
                              (* error occurred.     *)
```

Turbo Pascal for Windows

```
var  brd0 : integer;
   bdname : PChar;
bdname := 'GPIB0';            (* Factory default     *)
                              (* board name.         *)
brd0 := ibfind (bdname);
if brd0 < 0 then error;       (* If brd0 < 0, an     *)
                              (* error occurred.     *)
```

IBGTS IBGTS

Purpose: Go from Active Controller to Standby.

Format: `ibgts (ud,v)`

`ud` represents 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 (0), no shadow handshake or holdoff occurs.

The `ibgts` function makes the GPIB board go to the Controller Standby state and unasserts the ATN signal if the board is already the Active Controller. `ibgts` permits the GPIB Controller board to go to standby, allowing transfers between GPIB devices without intervention.

If the shadow handshake option is activated, the GPIB board participates in data handshake as an Acceptor without actually reading the data. The board monitors the transfers for the END message and holds off subsequent transfers. In this way, 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.

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.

Pascal

```
var cmd : cbuf;
cmd[1] := chr (UNL);          (* Unlisten            *)
cmd[2] := 'F';                (* talk address        *)
cmd[3] := '1';                (* listen address      *)
ibcmd (brd0,cmd,3);
ibgts (brd0,1);               (* Listen in           *)
                              (* continuous mode.    *)
```

Turbo Pascal for Windows

```
var cmd : PChar;
cmd := '?F1';
ibcmd (brd0,cmd^,3);          (* Unlisten            *)
                              (* talk address        *)
                              (* listen address.     *)
ibgts (brd0,1);               (* Listen in           *)
                              (* continuous mode.    *)
```

IBIST IBIST

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

Format: `ibist (ud,v)`

`ud` represents an interface board. If `v` is non-zero, the individual status bit is set. If `v` is zero (0), 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 one-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`.

Board Function Examples:

1. Set the individual status bit.

```
var v : integer;
v := 1;  (* Any non-zero value will do.            *)
ibist (brd0,v);
```

2. Clear the individual status bit.

```
var v : integer;
v := 0;
ibist (brd0,v);
```

IBLINES IBLINES

Purpose: Return the status of the GPIB control lines.

Format: `iblines (ud,clines)`

`ud` represents 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 is the same for both the low-order and upper byte, and 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 it can be monitored (indicated by a 1 in the appropriate bit position), then check the corresponding bit in the upper byte. If the bit in the upper byte is set (1), the corresponding control line is asserted. If the bit is clear (0), the control line is unasserted.

IBLINES (continued) IBLINES

Device/Board Function Example:

Test for Remote Enable (REN).

IBM/MS Pascal

```
var  brd0 : integer;
   bdname : cbuf;
   clines : integer;
bdname := 'GPIB0  ';
brd0 := ibfind (bdname);
if brd0 < 0 then error;
iblines (0,clines);
if (ibsta and ERR) <> 0 then error;
if (clines and 16#10) <> 16#10 then
    begin
       writeln ('GPIB board cannot monitor REN!');
       return;
    end;
if (clines and 16#1000) = 16#1000 then
    begin
       writeln ('REN is asserted.');
       return;
    end;
writeln ('REN is not asserted.');
```

IBLINES (continued) IBLINES

QuickPascal/Turbo Pascal

```
var  brd0 : integer;
   bdname : cbuf;
   clines : integer;
bdname := 'GPIB0  ';
brd0 := ibfind (bdname);
if brd0 < 0 then error;
iblines (0,clines);
if (ibsta and ERR) <> 0 then error;
if (clines and $10) <> $10 then
    begin
       writeln ('GPIB board cannot monitor REN!');
       halt;
    end;
if (clines and $1000) = $1000 then
    begin
       writeln ('REN is asserted.');
       halt;
    end;
writeln ('REN is not asserted.');
(* Exit to DOS                                    *)
```

Turbo Pascal for Windows

```
var  brd0 : integer;
   bdname : PChar;
   clines : integer;
bdname := 'GPIB0';
brd0 := ibfind (bdname);
if brd0 < 0 then error;
iblines (0,clines);
if (ibsta and ERR) <> 0 then error;
if (clines and $10) <> $10 then
    begin
       writeln ('GPIB board cannot monitor REN!');
       halt;
    end;
if (clines and $1000) = $1000 then
    begin
       writeln ('REN is asserted.');
       halt;
    end;
writeln ('REN is not asserted.');
```

IBLN IBLN

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

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

`ud` represents a board or device descriptor. `pad` (legal values are 0 to 30) is the primary 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. `sad` (legal values are hex 60 to 7e, `NO_SAD`, or `ALL_SAD`) is the secondary GPIB address of the device. The two special constants that can be used in place of a secondary address are defined in the declaration files as follows:

```
NO_SAD = 0
ALL_SAD = -1
```

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

Device Function Example:

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

IBM/MS Pascal

```
var ud,listen : integer;
       bdname : nbuf;
bdname := 'dev1   ';
ud := ibfind (bdname);
ibln  (ud,2,16#60,listen);
if (listen = 0) then
     begin
     (* Error: No device is at the address.        *)
     end;
```

IBLN (continued) IBLN

QuickPascal/Turbo Pascal

```
var ud,listen : integer;
       bdname : nbuf;
bdname := 'dev1   ';
ud := ibfind (bdname);
ibln (ud,2,$60,listen);
if (listen = 0) then
     begin
     (* Error: No device is at the address.        *)
     end;
```

Turbo Pascal for Windows

```
var ud,listen : integer;
ud := ibfind ('dev1');
ibln (ud,2,$60,listen);
if (listen = 0) then
     begin
     (* Error: No device is at the address.        *)
     end;
```

Board Function Example:

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

IBM/MS Pascal

```
var ud,listen : integer;
       bdname : nbuf;
bdname := 'gpib0  ';
ud := ibfind (bdname);
ibln  (ud,2,16#60,listen);
if (listen = 0) then
     begin
     (* Error: No device is at the address.        *)
     end;
```

IBLN (continued) IBLN

QuickPascal/Turbo Pascal

```
var ud,listen : integer;
       bdname : nbuf;
bdname := 'gpib0  ';
ud := ibfind (bdname);
ibln  (ud,2,$60,listen);
if (listen = 0) then
     begin
     (* Error: No device is at the address.        *)
     end;
```

Turbo Pascal for Windows

```
var ud,listen : integer;
ud := ibfind ('gpib0');
ibln  (ud,2,$60,listen);
if (listen = 0) then
     begin
     (* Error: No device is at the address.        *)
     end;
```

IBLOC IBLOC

Purpose: Go To Local state.

Format: `ibloc (ud)`

`ud` represents 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

If `ud` is a device, `ibloc` places the device indicated in local mode by calling `ibcmd` to send the following command sequence:

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

Other command bytes may 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), unless it is 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 if the computer is used as an instrument.

IBLOC (continued) IBLOC

Device Function Example:

Return the device `dvm` to local state.

```
ibloc (dvm);
```

Board Function Example:

Return the interface board `brd0` to local state.

```
ibloc (brd0);
```

IBONL IBONL

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

Format: `ibonl (ud,v)`

`ud` represents 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 (0), 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`.

```
var v : integer;
v := 0;
ibonl (plotter,v);
```

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

MS-DOS Pascal

```
var plotter : integer;
     bdname : nbuf;
bdname := 'plotter';    (* Device name assigned     *)
                        (* at configuration         *)
                        (* time.                    *)
plotter := ibfind (bdname);
(* ibonl with v non-zero is automatically          *)
(* performed as part of ibfind.                    *)
```

IBONL (continued) IBONL

Turbo Pascal for Windows

```
var plotter : integer;
     bdname : PChar;
bdname := 'plotter';    (* Device name assigned      *)
                        (* at configuration          *)
                        (* time.                     *)
plotter := ibfind (bdname);
(* ibonl with v non-zero is automatically           *)
(* performed as part of ibfind.                     *)
```

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

```
var v : integer;
v := 1;
ibonl (plotter,v);
```

Board Function Examples:

1. Disable the interface board `brd0`.

```
var v : integer;
v := 0;
ibonl (brd0,v);
```

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

MS-DOS Pascal

```
var brd0 : integer;
  bdname : nbuf;
bdname := 'GPIB0  ';    (* Board name assigned       *)
                        (* at configuration          *)
                        (* time.                     *)
brd0 := ibfind (bdname);
(* ibonl with v non-zero is automatically           *)
(* performed as part of ibfind.                     *)
```

IBONL (continued) IBONL

Turbo Pascal for Windows

```
var brd0 : integer;
  bdname : PChar;
bdname := 'GPIB0';      (* Board name assigned      *)
                        (* at configuration         *)
                        (* time.                    *)
brd0 := ibfind (bdname);
(* ibonl with v non-zero is automatically          *)
(* performed as part of ibfind.                    *)
```

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

```
var v : integer;
v := 1;
ibonl (brd0,v);
```

IBPAD IBPAD

Purpose: Change Primary Address.

Format: `ibpad (ud,v)`

`ud` represents a device or an interface board. `v` is the primary GPIB address.

`ibpad` is needed only to alter the configuration setting. 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.

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.

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`.

IBPAD (continued) IBPAD

Device Function Example:

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

IBM/MS Pascal

```
var v : integer;
v := 16#A;     (* Lower 5 bits of GPIB address.     *)
ibpad (plotter,v);
```

QuickPascal/Turbo Pascal/Turbo Pascal for Windows

```
var v : integer;
v := $A;       (* Lower 5 bits of GPIB address.     *)
ibpad (plotter,v);
```

Board Function Example:

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

IBM/MS Pascal

```
var v : integer;
v := 16#7;     (* Lower 5 bits of GPIB address.     *)
ibpad (brd0,v);
```

QuickPascal/Turbo Pascal/Turbo Pascal for Windows

```
var v : integer;
v := $7;       (* Lower 5 bits of GPIB address.     *)
ibpad (brd0,v);
```

IBPCT IBPCT

Purpose: Pass control.

Format: `ibpct (ud)`

`ud` represents 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 may be sent as necessary.

Device Function Example:

Pass control to the device `ibmxt`.

```
ibpct (ibmxt);
```

IBPPC IBPPC

Purpose: Parallel Poll Configure.

Format: `ibppc (ud,v)`

`ud` represents 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 (UNL)
- 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 of `v` 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 responds to a parallel poll by setting its Local Poll Enable (LPE) message to `v`.

Device Function Examples:

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

   IBM/MS Pascal

   ```
   var v : integer;
   v := 16#64;
   ibppc (dvm,v);
   ```

   QuickPascal/Turbo Pascal/Turbo Pascal for Windows

   ```
   var v : integer;
   v := $64;
   ibppc (dvm,v);
   ```

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

   IBM/MS Pascal

   ```
   var v : integer;
   v := 16#68;
   ibppc (dvm,v);
   ```

   QuickPascal/Turbo Pascal/Turbo Pascal for Windows

   ```
   var v : integer;
   v := $68;
   ibppc (dvm,v);
   ```

IBPPC (continued) IBPPC

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

IBM/MS Pascal

```
var v : integer;
v := 16#70;
ibppc (dvm,v);
```

QuickPascal/Turbo Pascal/Turbo Pascal for Windows

```
var v : integer;
v := $70;
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).

IBM/MS Pascal

```
var v : integer;
v := 16#64;
ibppc (brd0,v);
```

QuickPascal/Turbo Pascal/Turbo Pascal for Windows

```
var v : integer;
v := $64;
ibppc (brd0,v);
```

IBRD IBRD

Purpose: Read data from a device to a string.

Format: `ibrd (ud,rd,cnt)`

`ud` represents a device or an interface board. The pre-allocated array `rd` identifies the storage buffer for data. `cnt` specifies the number of bytes to be read from the GPIB. `cnt` is of type `integer4` in Microsoft Pascal and of type `longINT` in QuickPascal, Turbo Pascal, and Turbo Pascal for Windows. `cnt` is of type `integer` in IBM Pascal.

`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).

The 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, `ibcntl` is the number of bytes read, `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.

IBRD (continued) IBRD

Board IBRD Function

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

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.

Pascal

```
var ud : integer;
    rd : cbuf;
ud := ibdev (0,10,0,15,1,0);
ibrd (ud,rd,100);
```

Turbo Pascal for Windows

```
var ud : integer;
    rd : array[0..99] of char;
ud := ibdev (0,10,0,15,1,0);
ibrd (ud,rd,100);
```

IBRD (continued) IBRD

Board Function Examples:

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

IBM/MS Pascal

```
var cmd : cbuf;
cmd[1] := chr(UNL);       (* Unlisten.                *)
cmd[2] := chr(16#4C);     (* Device talk address.     *)
cmd[3] := chr(16#20);     (* GPIB board listen        *)
                          (* address.                 *)
ibcmd (brd0,cmd,3);
ibrd (brd0,rd,56);      (* Check ibsta to see on      *)
                        (* what the read terminated:  *)
                        (* CMPL, END, TIMO, or ERR.   *)
                        (* Data is stored in rd       *)
                        (* Unaddress the              *)
                        (* Talker and Listener.       *)
cmd[1] := chr(UNT);
cmd[2] := chr(UNL);
ibcmd (brd0,cmd,2);
```

QuickPascal/Turbo Pascal

```
var cmd : cbuf;
cmd[1] := chr(UNL);       (* Unlisten.                *)
cmd[2] := chr($4C);       (* Device talk address      *)
cmd[3] := chr($20);       (* GPIB board listen        *)
                          (* address.                 *)
ibcmd (brd0,cmd,3);
ibrd (brd0,rd,56);      (* Check ibsta to see on      *)
                        (* what the read terminated:  *)
                        (* CMPL, END, TIMO, or ERR.   *)
                        (* Data is stored in rd       *)
                        (* Unaddress the              *)
                        (* Talker and Listener.       *)
cmd[1] := chr(UNT);
cmd[2] := chr(UNL);
ibcmd (brd0,cmd,2);
```

IBRD (continued) IBRD

Turbo Pascal for Windows

```
var rd : array[0..55] of char;
  cmd : PChar;
(* Unlisten.Device talk address                    *)
(* GPIB board listen address                       *)
cmd := '?'#4C#20;
ibcmd (brd0,cmd^,3);
ibrd (brd0,rd,56);
(* Check ibsta to see on what the read terminated: *)
(* CMPL, END, TIMO, or ERR.  Data is stored in rd  *)
(* Unaddress the Talker and Listener.              *)
cmd := '_?';
ibcmd (brd0,cmd^,2);
```

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: `ibrda (ud,rd,cnt)`

`ud` represents a device or an interface board. The pre-allocated array `rd` identifies the storage buffer for instruction bytes that are read from the GPIB. `cnt` specifies the number of bytes to be read from the GPIB. `cnt` is of type `integer4` in Microsoft Pascal and of type `longINT` in QuickPascal, Turbo Pascal, and Turbo Pascal for Windows. `cnt` is of type `integer` in IBM Pascal.

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

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.

Device Function Example:

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

Pascal

```
var rd : cbuf;
    mask : word;
ibrda (tape,rd,56);     (* Perform device read.     *)
mask := CMPL or TIMO;
(* Perform other processing here.  Then wait       *)
(* for I/O completion or a timeout.                *)
ibwait (tape,mask);
(* ibsta shows how the read terminated: on         *)
(* CMPL, END, TIMO, or ERR.                        *)
```

IBRDA (continued) IBRDA

Turbo Pascal for Windows

```
var   rd : array[0..55] of char;
    mask : word;
(* Read into null-terminated string                *)
ibrda (tape,rd,56);     (* Perform device read.     *)
mask := CMPL or TIMO;
(* Perform other processing here.  Then wait       *)
(* for I/O completion or a timeout.                *)
ibwait (tape,mask);
(* ibsta shows how the read terminated: on         *)
(* CMPL, END, TIMO, or ERR.                        *)
```

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).

IBM/MS Pascal

```
var  cmd,rd  : cbuf;
        mask : word;
(* Perform addressing in preparation for a         *)
(* board read.                                     *)
cmd[1] := chr(16#3F);
cmd[2] := chr(16#20);
cmd[3] := chr(16#4C);
ibcmd (brd0,cmd,3);
ibrda (brd0,rd,56);
eventtst;   (* Unspecified routine to test         *)
            (* and process the high priority       *)
            (* event.                              *)
v := TIMO + CMPL;
ibwait (brd0,v);   (* Wait for I/O to complete.    *)
(* ibsta shows how the read terminated: on         *)
(* CMPL, END, TIMO, or ERR.                        *)
(* Data is stored in rd.                           *)
(* Unaddress the Talker and Listener.              *)
cmd[1] := chr(16#5F);
cmd[2] := chr(16#3F);
ibcmd (brd0,cmd,2);
```

IBRDA (continued) IBRDA

QuickPascal/Turbo Pascal

```
var cmd,rd  : cbuf;
      mask  : word;
(* Perform addressing in preparation for a         *)
(* board read.                                     *)
cmd[1] := chr($3F);
cmd[2] := chr($20);
cmd[3] := chr($4C);
ibcmd (brd0,cmd,3);
ibrda (brd0,rd,56);
eventtst; (* Unspecified routine to test and       *)
          (* process the high priority event.     *)
v := TIMO + CMPL;
ibwait (brd0,v);   (* Wait for I/O to complete.    *)
(* ibsta shows how the read terminated: on         *)
(* CMPL, END, TIMO, or ERR.                        *)
(* Data is stored in rd.                           *)
(* Unaddress the Talker and Listener.              *)
cmd[1] := chr($5F);
cmd[2] := chr($3F);
ibcmd (brd0,cmd,2);
```

Turbo Pascal for Windows

```
var cmd : PChar;
    rd  : array[0..55] of char;
   mask : word;
(* Perform addressing in preparation for a         *)
(* board read.                                     *)
cmd := #3F#20#4C;
ibcmd (brd0,cmd^,3);
(* Read into null-terminated string                *)
ibrda (brd0,rd,56);
eventtst; (* Unspecified routine to test and       *)
          (* process the high priority event.     *)
v := TIMO + CMPL;
ibwait (brd0,v);   (* Wait for I/O to complete.    *)
(* ibsta shows how the read terminated: on         *)
(* CMPL, END, TIMO, or ERR.                        *)
(* Data is stored in rd.                           *)
(* Unaddress the Talker and Listener.              *)
cmd := #5F#3F;
ibcmd (brd0,cmd^,2);
```

IBRDA (continued) IBRDA

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.*

IBRDF IBRDF

Purpose: Read data from GPIB into file.

Format: `ibrdf (ud,flname)`

`ud` represents 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. For MS-DOS Pascal `flname` is of type `flbuf` as defined in the header files `DECL.PAS`, `QPDECL.PAS`, and `TPDECL.PAS`. For Turbo Pascal for Windows, `flname` is of type `PChar`.

`ibrdf` automatically opens the file `flname` 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, `ibcntl` is the number of instruction bytes read. `ibcnt` is the 16-bit representation of the number of bytes read.

When the device `ibrdf` function returns, `ibsta` holds the latest device status, `ibcntl` is the number of instruction bytes read, `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 `ibrd` function performs board functions 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 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`.

MS-DOS Pascal

```
var   flname : flbuf;
(* Perform device read. flname has been blank-filled.*)
flname[1] := 'B';
flname[2] := ':';
flname[3] := 'R';
flname[4] := 'D';
flname[5] := 'G';
flname[6] := 'S';
ibrdf  (rdr,flname);
(* ibsta and ibcnt show the results of the         *)
(* read operation.                                 *)
```

Turbo Pascal for Windows

```
var   flname : PChar;
flname := 'B:RDGS';
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).

MS-DOS Pascal

```
var   cmd : cbuf
   flname : flbuf;
(* Perform addressing in preparation for a          *)
(* board read.                                      *)
cmd[1] := chr(UNL);
cmd[2] := chr('L');           (* TAD                *)
cmd[3] := chr(' ');           (* MLA                *)
ibcmd (brd0,cmd,3);
(* flname has been blank-filled.                    *)
flname[1] := 'R';
flname[2] := 'D';
flname[3] := 'G';
flname[4] := 'S';
ibrdf (brd0,flname);
(* ibsta and ibcnt show the results of the          *)
(* completed operation.                             *)
```

Turbo Pascal for Windows

```
var   cmd,flname : PChar;
(* Perform addressing in preparation for a          *)
(* board read.                                      *)
cmd := '?L';
ibcmd (brd0,cmd^,3);
flname := 'RDGS';
ibrdf (brd0,flname);
(* ibsta and ibcnt show the results of the          *)
(* completed operation.                             *)
```

IBRDI IBRDI

Purpose: Read data to integer array.

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

`ibrdi` is not available in Turbo Pascal or Turbo Pascal for Windows because the `ibrd` call is adequate for receiving data into any type of buffer. IBM/MS Pascal and QuickPascal, which have more rigid typing rules, require different procedure declarations for array buffers of different types.

`ud` represents a device or an interface board. `iarr` is of type `ibuf` as defined in the header files `DECL.PAS` and `QPDECL.PAS`. `cnt` specifies the maximum number of bytes to be read. `cnt` is of type `integer4` in Microsoft Pascal, `longINT` in QuickPascal, and `integer` in IBM Pascal.

`ibrdi` is similar to the `ibrd` function, which reads into a character array. As the data is read, each byte pair is treated as an integer and stored in `iarr`. But unlike `ibrd`, `ibrdi` stores the data directly into an integer array. No integer conversion of the data is needed for arithmetic operations.

Device Function Example:

Read 512 bytes of data from the device `tape` and store in the integer array `rd`.

IBM/MS Pascal/QuickPascal

```
var tape : integer;
      rd : ibuf;
tape := ibdev (0,6,0,14,1,0);
ibrdi (tape,rd,512);
```

IBRDI (continued) IBRDI

Board Function Examples:

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

   IBM/MS Pascal/QuickPascal

```
var    bd : integer;
   bdname : nbuf;
      cmd : cbuf;
       rd : ibuf;
bdname := 'GPIB0  ';
bd := ibfind (bdname);
cmd[1] := '?';                (* UNL                *)
cmd[2] := ' ';                (* MLA                *)
cmd[3] := 'L';                (* MTA                *)
ibcmd (bd,cmd,3);
ibrdi (bd,rd,56);
(* ibsta shows how the read terminated:            *)
(* on CMPL, END, TIMO, or ERR.                     *)
(* 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*.

IBRDIA IBRDIA

Purpose: Read data asynchronously to integer array.

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

`ibrdia` is not available in Turbo Pascal or Turbo Pascal for Windows because the `ibrda` call is adequate for receiving data into any type of buffer. IBM/MS Pascal and QuickPascal, which have more rigid typing rules, require different procedure declarations for array buffers of different types.

`ud` represents a device or an interface board. `iarr` is of type `ibuf` as defined in the header files `DECL.PAS` and `QPDECL.PAS`. `cnt` is the maximum number of bytes to be read. `cnt` is of type `integer4` in Microsoft Pascal, `longINT` in QuickPascal, and `integer` in IBM Pascal.

`ibrdia` is similar to the `ibrda` function, which reads into a character string variable. As the data is read, each byte pair is treated as an integer and stored in `iarr`. But unlike `ibrda`, `ibrdia` stores the data directly into an integer array. No integer conversion of the data is needed for arithmetic operations.

Device Function Example:

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

IBM/MS Pascal

```
var   rd : ibuf;
    mask : word;
(* Perform device read.                              *)
ibrdia (tape,rd,56);
(* Perform other processing here.  Then wait        *)
(* for I/O completion or a timeout.                 *)
mask := 16#4100;              (* TIMO CMPL          *)
ibwait (tape,mask);
(* ibsta shows how the read terminated:  on         *)
(* CMPL, END, TIMO, or ERR.                         *)
(* Data is stored in rd.                            *)
if (ibsta and ERR) <> 0 then error;
```

IBRDIA (continued) IBRDIA

QuickPascal

```
var   rd : ibuf;
    mask : word;
(* Perform device read.*)
ibrdia (tape,rd,56);
(* Perform other processing here.  Then wait       *)
(* for I/O completion or a timeout.                *)
mask := $4100;                (* TIMO CMPL        *)
ibwait (tape,mask);
(* ibsta shows how the read terminated:  on        *)
(* CMPL, END, TIMO, or ERR.                        *)
(* Data is stored in rd.                           *)
if (ibsta and ERR) <> 0 then error;
```

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.)

IBM/MS Pascal

```
var cmd : cbuf;
     rd : ibuf;
   mask : word;
(* Perform addressing in preparation for           *)
(* board read.                                     *)
cmd[1] := '?';                (* UNL              *)
cmd[2] := ' ';                (* MLA              *)
cmd[3] := 'L';                (* TAD              *)
ibcmd (bd,cmd,3);
ibrdia (bd,rd,56);
(* Perform other processing here.  Then wait       *)
(* for I/O completion or a timeout.                *)
mask := 16#4100;              (* TIMO CMPL        *)
ibwait (bd,mask);
(* ibsta shows how the read terminated:  on        *)
(* CMPL, END, TIMO, or ERR.                        *)
(* Data is stored in rd.                           *)
if (ibsta and ERR) <> 0 then error;
```

IBRDIA (continued) IBRDIA

QuickPascal

```
var cmd : cbuf;
     rd : ibuf;
   mask : word;
(* Perform addressing in preparation for           *)
(* board read.                                     *)
cmd[1] := '?';                (* UNL               *)
cmd[2] := ' ';                (* MLA               *)
cmd[3] := 'L';                (* TAD               *)
ibcmd (bd,cmd,3);
ibrdia (bd,rd,56);
(* Perform other processing here.  Then wait      *)
(* for I/O completion or a timeout.                *)
mask := $4100;                (* TIMO CMPL         *)
ibwait (bd,mask);
(* ibsta shows how the read terminated:  on        *)
(* CMPL, END, TIMO, or ERR.                        *)
(* Data is stored in rd.                           *)
if (ibsta and ERR) <> 0 then error;
```

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: `ibrpp (ud,ppr)`

`ud` represents 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 function `ibrpp` 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 3-6.

IBRPP (continued) IBRPP

Table 3-6. 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 appropriate declaration file.

Device Function Example:

Remotely configure the device `lcrmtr` to respond positively on DIO3 if its individual status bit is 1, and then parallel poll all configured devices.

IBM/MS Pascal

```
var ppc,ppr : integer;
ppc := 16#6A;
ibppc (lcrmtr,ppc);
ibrpp (lcrmtr,ppr);
```

QuickPascal/Turbo Pascal/Turbo Pascal for Windows

```
var ppc,ppr : integer;
ppc := $6A;
ibppc (lcrmtr,ppc);
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.

   IBM/MS Pascal

```
var ppr : integer;
    cmd : cbuf;
cmd[1] := chr(16#23);    (* device listen address   *)
cmd[2] := chr(PPC);
cmd[3] := chr(PPE + S + 2); (* Send PPR3 if ist = 1 *)
cmd[4] := chr(UNL);
ibcmd (brd0,cmd,4);
ibrpp (brd0,ppr);    (* PPR is returned in ppr.     *)
```

   QuickPascal/Turbo Pascal/Turbo Pascal for Windows

```
var ppr : integer;
    cmd : cbuf;
cmd[1] := chr($23);     (* device listen address    *)
cmd[2] := chr(PPC);
cmd[3] := chr(PPE + S + 2); (* Send PPR3 if ist = 1 *)
cmd[4] := chr(UNL);
ibcmd (brd0,cmd,4);
ibrpp (brd0,ppr);    (* PPR is returned in ppr.     *)
```

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

```
var cmd : cbuf;
cmd[1] := chr(PPU);
ibcmd (brd0,cmd,1);
```

IBRSC IBRSC

Purpose: Request or release System Control.

Format: `ibrsc (ud,v)`

`ud` represents 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, but in some 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 can 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.

Board Function Example:

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

```
var v : integer;
v := 1;  (* Any non-zero value will do.              *)
ibrsc (brd0,v);
```

IBRSP IBRSP

Purpose: Return serial poll byte.

Format: `ibrsp (ud,spr)`

`ud` represents 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. Controller 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.

Device Function Example:

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

```
var spr : integer;
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: `ibrsv (ud,v)`

`ud` represents 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`.

Board Function Examples:

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

   IBM/MS Pascal

   ```
   var  v : integer;
   v := 16#41 or 1;            (* assert srq          *)
   ibrsv (brd0,v);
   ```

   QuickPascal/Turbo Pascal/Turbo Pascal for Windows

   ```
   var  v : integer;
   v := $41 or 1;
   ibrsv (brd0,v);
   ```

IBRSV (continued) IBRSV

2. Change the status byte without requesting service.

IBM/MS Pascal

```
var v : integer;
v := 16#23;
ibrsv (brd0,v);
```

QuickPascal/Turbo Pascal/Turbo Pascal for Windows

```
var v : integer;
v := $23;
ibrsv (brd0,v);
```

IBSAD IBSAD

Purpose: Change or disable Secondary Address.

Format: `ibsad (ud,v)`

`ud` represents 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.

Device Function Examples:

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

   IBM/MS Pascal

   ```
   var v : integer;
   v := 16#6A;
   ibsad (plotter,v);
   ```

IBSAD (continued) IBSAD

QuickPascal/Turbo Pascal/Turbo Pascal for Windows

```
var v : integer;
v := $6A;
ibsad (plotter,v);
```

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

```
var v : integer;
v := 0;          (* 0 or hex 7F can be used.        *)
ibsad (dvm,v);
```

Board Function Examples:

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

IBM/MS Pascal

```
var v : integer;
v := 16#6A;
ibsad (brd0,v);
```

QuickPascal/Turbo Pascal/Turbo Pascal for Windows

```
var v : integer;
v := $6A;
ibsad (brd0,v);
```

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

```
var v : integer;
v := 0;            (* 0 or hex 7F can be used.      *)
ibsad (brd0,v);
```

IBSIC IBSIC

Purpose: Send interface clear for 100 µsec.

Format: `ibsic (ud)`

`ud` represents 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 if the GPIB board is System Controller. 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.

Board Function Example:

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

```
ibsic (brd0);
```

IBSRE IBSRE

Purpose: Set or clear the Remote Enable line.

Format: `ibsre (ud,v)`

`ud` represents 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. 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 REN state is stored in `iberr`.

Board Function Examples:

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

   MS-DOS Pascal

```
var cmd : cbuf;
ibsre (brd0,1);  (* Any non-zero value will do.     *)
cmd[1] := '#';        (* device listen address      *)
ibcmd (brd0,cmd,1);
```

   Turbo Pascal for Windows

```
var cmd : PChar;
ibsre (brd0,1);  (* Any non-zero value will do.     *)
cmd := '#';           (* device listen address      *)
ibcmd (brd0,cmd^,1);
```

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.

IBM/MS Pascal

```
var cmd : cbuf;
cmd[1] := chr(LLO);     (* or:                          *)
                        (* cmd[1] = chr(16#23);         *)
ibcmd (brd0,cmd,1);     (* cmd[2] = chr(LLO);           *)
                        (* ibcmd (brd0,cmd,2);          *)
```

QuickPascal/Turbo Pascal

```
var cmd : cbuf;
cmd[1] := chr(LLO);     (* or:                          *)
                        (* cmd[1] = chr($23);           *)
ibcmd (brd0,cmd,1);     (* cmd[2] = chr(LLO);           *)
                        (* ibcmd (brd0,cmd,2);          *)
```

Turbo Pascal for Windows

```
var cmd : PChar;
cmd := #11;             (* or:  cmd = #23#11;           *)
                        (*      ibcmd(brd0,cmd^,2);     *)
ibcmd (brd0,cmd^,1);
```

3. Return all devices to local mode.

```
var v : integer;
v := 0;                 (* Set REN to false.            *)
ibsre (brd0,v);
```

IBSRQ IBSRQ

Purpose: Register an SRQ "interrupt routine."

Format:

IBM Pascal

```
ibsrq (func);
```

MS Pascal

```
ibsrq (ads func);
```

QuickPascal/Turbo Pascal

```
ibsrq (@ func);
```

This function establishes the Pascal, QuickPascal, or Turbo Pascal routine `func` as the procedure to be called whenever the driver notices the SRQI bit set (1) in the status word (`ibsta`) of a GPIB interface board. The check for SRQI is made after each call to the driver. If SRQI is set, `func` will be called before control is returned to the application program. SRQ servicing is turned off if `ibsrq` is called with the `ibnil` procedure. `ibnil` is declared in the header files `DECL.PAS`, `QPDECL.PAS`, and `TPDECL.PAS`. `ibnil` is defined in the language interface files `PIB.OBJ` and `TPIB.OBJ`. Turbo Pascal for Windows does not support `IBSRQ`.

Note: Disable automatic serial polling if you use `ibsrq`.

IBM/MS Pascal

The procedure to be called must be a separate Pascal module. The procedure must be declared external (`extern`) in the main program. Do not establish a nested procedure as a procedure to be called. The actual call syntax is different for Microsoft Pascal and IBM Pascal.

IBSRQ (continued) IBSRQ

QuickPascal/Turbo Pascal

The procedure to be called is defined in the main program. Nested procedures are not allowed to be called for SRQ servicing. The heading of the procedure to be called is different for Microsoft Pascal and Turbo Pascal. The Turbo Pascal heading must include the directive `far`.

For example:

QuickPascal: `procedure srqservice;`

Turbo Pascal: `procedure srqservice; far;`

Example:

Establish `srqservice` as the procedure to call SRQ servicing.

MS Pascal

```
procedure srqservice; extern;
var          bd : integer;
   dvm [public] : integer;
   spr [public] : integer;
         udname : nbuf;
begin
   udname := 'gpib0  ';
   bd := ibfind (udname);
   (* Disable autopolling.                          *)
   ibconfig (bd,IbcAUTOPOLL,0);
   udname := 'DEV4  ';
   dvm := ibfind (udname);
   ibsrq (ads srqservice);
end.

module separate pascal module;
procedure ibrsp (dvm : integer; spr : integer);
extern;
procedure srqservice;
var   dvm [extern] : integer;
      spr [extern] : integer;
```

IBSRQ (continued) IBSRQ

```
begin
   ibrsp (dvm,spr);
   (* Analyze the response byte here.              *)
end;
end.
```

IBM Pascal

```
procedure srqservice; extern;
var          bd : integer;
   dvm [public] : integer;
   spr [public] : integer;
         udname : nbuf;
begin
   udname := 'gpib0  ';
   bd := ibfind (udname);
   (* Disable autopolling.                         *)
   ibconfig (bd,IbcAUTOPOLL,0);
   udname := 'DEV4  ';
   dvm := ibfind (udname);
   ibsrq (srqroutine);
end.

module separate pascal module;
procedure ibrsp (dvm : integer; spr : integer);
extern;
procedure srqservice;
var   dvm [extern] : integer;
      spr [extern] : integer;
begin
   ibrsp (dvm,spr);
   (* Analyze the response byte here.              *)
end;
end.
```

IBSRQ (continued) IBSRQ

QuickPascal

```
var bd,dvm : integer;
    udname : nbuf;
procedure srqservice;
var spr;
begin
   ibrsp (dvm,spr);
   (* Analyze the response here.                    *)
end;

begin     (* Main                                   *)
   udname := 'GPIB0  ';
   bd := ibfind (udname);
   (* Disable autopolling                           *)
   ibconfig (bd,IbcAUTOPOLL,0);
   udname := 'DEV4   ';
   dvm := ibfind (udname);
   ibsrq (@ srqservice);
end.
```

Turbo Pascal

```
var bd,dvm : integer;
    udname : nbuf;
procedure srqservice; far;
var spr;
begin
   ibrsp (dvm,spr);
   (* Analyze the response here.                    *)
end;

begin     (* Main                                   *)
   udname := 'GPIB0  ';
   bd := ibfind (udname);
   (* Disable autopolling                           *)
   ibconfig (bd,IbcAUTOPOLL,0);
   udname := 'DEV4   ';
   dvm := ibfind (udname);
   ibsrq (@ srqservice);
end.
```

IBSTOP IBSTOP

Purpose: Abort asynchronous operation.

Format: `ibstop (ud)`

`ud` represents 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 asynchronous 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`.

```
ibstop (rdr);
```

Board Function Example:

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

```
ibstop (brd0);
```

IBTMO IBTMO

Purpose: Change or disable time limit.

Format: `ibtmo (ud,v)`

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

Table 3-7. 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

Note: If `v` is zero (0), no limit is in effect.

IBTMO (continued) IBTMO

`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`.

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.

IBTMO (continued) IBTMO

Device Function Example:

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

MS-DOS Pascal

```
var   tape : integer;
   devname : nbuf;
devname := 'DEV9   ';
tape := ibfind (devname);
ibtmo (tape,10);
```

Turbo Pascal for Windows

```
var   tape : integer;
   devname : PChar;
devname := 'DEV9';
tape := ibfind (devname);
ibtmo (tape,10);
```

Board Function Example:

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

```
ibtmo (brd0,7);
```

IBTRAP IBTRAP

Purpose: Alter Applications Monitor trap and display modes.

Format: `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 values are listed in Table 3-8.

Table 3-8. 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 a call to `ibtrap`, the ERR bit of `ibsta` is set and `iberr` is one of the values listed in Table 3-9. Otherwise, `iberr` contains the previous `mask` value.

Table 3-9. IBTRAP Errors

Value	Explanation
ECAP	Applications Monitor not installed.
EARG	Invalid monitor mode.

Refer to Appendix B, *Applications Monitor*, for more information. Note that the Applications Monitor can only be used with NI-488.2 for MS-DOS.

IBTRAP (continued) IBTRAP

Device Function Example:

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

```
var mask : word;
    mode : integer;
mask := SRQI or CMPL;       (* mask is hex 1100    *)
mode := 3;
ibtrap (mask,mode);
```

IBTRG IBTRG

Purpose: Trigger selected device.

Format: `ibtrg (ud)`

`ud` represents 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 (UNL)
- Listen address of the device
- Secondary address of the device, if applicable
- Group Execute Trigger (GET)

Other command bytes may be sent as necessary.

Device Function Example:

Trigger the device `analyz`.

```
ibtrg (analyz);
```

IBWAIT IBWAIT

Purpose: Wait for selected event.

Format: `ibwait (ud,mask)`

`ud` represents 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 3-10. `ibwait` also updates `ibsta`.

Table 3-10. 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	Asynchronous 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

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. The responses are saved 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.

```
var  mask : word;
mask := RQS;            (* mask is hex 800          *)
ibwait (logger,mask);
```

IBWAIT (continued) IBWAIT

Board Function Examples:

1. Wait for a service request or a timeout.

```
var  mask : word;
mask := SRQI + TIMO;      (* mask is hex 5000       *)
ibwait (brd0,mask);
(* Check ibsta here to see which occurred.          *)
```

2. Update the current status for `ibsta`.

```
var  mask : word;
mask := 0;
ibwait (brd0,mask);
```

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

```
var  mask : word;
mask := CIC;              (* mask is hex 20         *)
ibwait (brd0,mask);
```

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

```
var  mask : word;
mask := TACS + LACS;      (* mask is hex 0C         *)
ibwait (brd0,mask);
```

IBWRT IBWRT

Purpose: Write data from string.

Format: `ibwrt (ud,wrt,cnt)`

`ud` represents a device or an interface board. `wrt` is a pre-allocated array containing the data to be sent over the GPIB. `cnt` specifies the number of bytes to be sent over the GPIB. `cnt` is of type `integer4` in Microsoft Pascal, and of type `longINT` in QuickPascal and Turbo Pascal. `cnt` is of type `integer` in IBM Pascal.

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.

After termination, `ibcntl` is the number of bytes read. `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, `ibcntl` is the actual number of instruction bytes written to the device, `ibcnt` is the 16-bit representation of the number of instruction 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.

IBWRT (continued) IBWRT

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 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.

If the access board is CIC, `ibcmd` must be called prior to `ibwrt` to address the device to listen and the board to talk. 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 instruction 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 Function Example 2*.

Device Function Examples:

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

   Pascal

```
var   wrt : cbuf;
wrt[1] := 'F';
wrt[2] := '3';
wrt[3] := 'R';
wrt[4] := '1';
wrt[5] := 'P';
wrt[6] := '2';
wrt[7] := 'X';
wrt[8] := '5';
wrt[9] := 'G';
wrt[10] := '0';
ibwrt (dvm,wrt,10);
```

IBWRT (continued) IBWRT

Turbo Pascal for Windows

```
var   wrt : PChar;
wrt := 'F3R1P2X5G0';
ibwrt (dvm,wrt^,10);
```

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.

IBM/MS Pascal

```
var  wrt : cbuf;
wrt[1] := 'I';
wrt[2] := 'P';
wrt[3] := '2';
wrt[4] := 'X';
wrt[5] := '5';
wrt[6] := chr(16#0D);
wrt[7] := chr(16#0A);
ibwrt (ptr,wrt,7);
```

QuickPascal/Turbo Pascal

```
var  wrt : cbuf;
wrt[1] := 'I';
wrt[2] := 'P';
wrt[3] := '2';
wrt[4] := 'X';
wrt[5] := '5';
wrt[6] := chr($0D);
wrt[7] := chr($0A);
ibwrt (ptr,wrt,7);
```

Turbo Pascal for Windows

```
var  wrt : PChar;
wrt := 'IP2X5'#0D#0A;
ibwrt (ptr,wrt^,7);
```

IBWRT (continued) IBWRT

Board Function Example:

Write 10 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 @).

IBM/MS Pascal

```
var   cmd : cbuf;
      wrt : cbuf;
cmd[1] := chr(UNL);       (* Unlisten.               *)
cmd[2] := chr(16#40);     (* PC talk address.        *)
cmd[3] := chr(16#2F);     (* Device listen address   *)
ibcmd (brd0,cmd,3);
wrt[1] := 'F'             (* Ten instruction bytes   *)
wrt[2] := '3';
wrt[3] := 'R';
wrt[4] := '1';
wrt[5] := 'P';
wrt[6] := '2';
wrt[7] := 'X';
wrt[8] := '5';
wrt[9] := 'G';
wrt[10] := '0';
ibwrt (brd0,wrt,10);
```

IBWRT (continued) IBWRT

QuickPascal/Turbo Pascal

```
var   cmd : cbuf;
      wrt : cbuf;
cmd[1] := chr(UNL);       (* Unlisten.              *)
cmd[2] := chr($40);       (* PC talk address        *)
cmd[3] := chr($2F);       (* device listen address  *)
ibcmd (brd0,cmd,3);
wrt[1] := 'F';            (* Ten instruction bytes  *)
wrt[2] := '3';
wrt[3] := 'R';
wrt[4] := '1';
wrt[5] := 'P';
wrt[6] := '2';
wrt[7] := 'X';
wrt[8] := '5';
wrt[9] := 'G';
wrt[10] := '0';
ibwrt (brd0,wrt,10);
```

Turbo Pascal for Windows

```
var   cmd,wrt : PChar;
(* Unlisten.  PC talk address.                    *)
(* device listen address.                         *)
cmd := '?'#40#2F;
ibcmd (brd0,cmd^,3);
wrt := 'F3R1P2X5G0';      (* Ten instruction bytes  *)
ibwrt (brd0,wrt^,10);
```

IBWRTA IBWRTA

Purpose: Write data asynchronously from string.

Format: `ibwrta (ud,wrt,cnt)`

`ud` represents a device or an interface board. `wrt` is a pre-allocated array containing the data to be sent over the GPIB. `cnt` specifies the number of bytes to be sent over the GPIB. `cnt` is of type `integer4` in Microsoft Pascal, and of type `longINT` in QuickPascal and Turbo Pascal. `cnt` is of type `integer` in IBM Pascal.

`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 NI-488 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 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.

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. 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. 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.

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

IBWRTA (continued) IBWRTA

Device Function Example:

Write 10 instruction bytes to the device `dvm` while performing other processing.

Pascal

```
var   wrt : cbuf;
     mask : word;
wrt[1] := 'F';          (* Ten instruction bytes    *)
wrt[2] := '3';
wrt[3] := 'R';
wrt[4] := '1';
wrt[5] := 'P';
wrt[6] := '2';
wrt[7] := 'X';
wrt[8] := '5';
wrt[9] := 'G';
wrt[10] := '0';
ibwrta (dvm,wrt,10);
mask := CMPL or TIMO;
(* Perform other processing here, then wait        *)
(* for I/O completion or a timeout.                *)
ibwait (dvm,mask);
(* ibsta indicates how the write terminated:       *)
(* on CMPL, END, TIMO, or ERR.                     *)
```

Turbo Pascal for Windows

```
var   wrt : PChar;
     mask : word;
wrt := 'F3R1P2X5G0';
ibwrta (dvm,wrt^,10);
mask := CMPL or TIMO;
(* Perform other processing here, then wait        *)
(* for I/O completion or a timeout.                *)
ibwait (dvm,mask);
(* ibsta indicates how the write terminated:       *)
(* on CMPL, END, TIMO, or ERR.                     *)
```

IBWRTA (continued) IBWRTA

Board Function Example:

Write 10 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 @).

IBM/MS Pascal

```
var  cmd : cbuf;
     wrt : cbuf;
    mask : word;
(* Perform addressing in preparation for           *)
(* board write.                                    *)
cmd[1] := chr(16#3F);         (* Unlisten          *)
cmd[2] := chr(16#40);         (* talk address      *)
cmd[3] := chr(16#2F);         (* listen address    *)
ibcmd (brd0,cmd,3);
(* Perform board asynchronous write.              *)
wrt[1] := 'F';
wrt[2] := '3';
wrt[3] := 'R';
wrt[4] := '1';
wrt[5] := 'P';
wrt[6] := '2';
wrt[7] := 'X';
wrt[8] := '5';
wrt[9] := 'G';
wrt[10] := '0';
ibwrta (brd0,wrt,10);
(* Perform other processing here.  Then wait      *)
(* for I/O completion or a timeout.               *)
mask := CMPL or TIMO;
ibwait (brd0,mask);
(* Unaddress the Talker and Listener.             *)
cmd[1] := chr(UNT);
cmd[2] := chr(UNL);
ibcmd (brd0,cmd,2);
```

IBWRTA (continued) IBWRTA

QuickPascal/Turbo Pascal

```
var  cmd : cbuf;
     wrt : cbuf;
    mask : word;
(* Perform addressing in preparation for           *)
(* board write.                                    *)
cmd[1] := chr($3F);    (* Unlisten                 *)
cmd[2] := chr($40);    (* talk address             *)
cmd[3] := chr($2F);    (* listen address           *)
ibcmd (brd0,cmd,3);
(* Perform board asynchronous write.               *)
wrt[1] := 'F';
wrt[2] := '3';
wrt[3] := 'R';
wrt[4] := '1';
wrt[5] := 'P';
wrt[6] := '2';
wrt[7] := 'X';
wrt[8] := '5';
wrt[9] := 'G';
wrt[10] := '0';
ibwrta (brd0,wrt,10);
(* Perform other processing here.  Then wait       *)
(* for I/O completion or a timeout.                *)
mask := CMPL or TIMO;
ibwait (brd0,mask);
(* Unaddress the Talker and Listener.              *)
cmd[1] := chr(UNT);
cmd[2] := chr(UNL);
ibcmd (brd0,cmd,2);
```

IBWRTA (continued) IBWRTA

Turbo Pascal for Windows

```
var  cmd,wrt : PChar;
       mask : word;
(* Perform addressing in preparation for          *)
(* board write.                                   *)
cmd := #3F#40#2F;
ibcmd (brd0,cmd^,3);
(* Perform board asynchronous write.              *)
wrt := 'F3R1P2X5G0';
ibwrta (brd0,wrt^,10);
(* Perform other processing here.  Then wait      *)
(* for I/O completion or a timeout.               *)
mask := CMPL or TIMO;
ibwait (brd0,mask);
(* Unaddress the Talker and Listener.             *)
cmd := '_?';
ibcmd (brd0,cmd^,2);
```

IBWRTF IBWRTF

Purpose: Write data from file.

Format: `ibwrtf (ud,flname)`

`ud` represents a device or an interface board. `flname` is the filename from which the data is written to the GPIB. `flname` may be up to 50 characters long, including a drive and path designation. For MS-DOS Pascal, `flname` is of type `flbuf` as defined in the declaration files `DECL.PAS`, `QPDECL.PAS`, and `TPDECL.PAS`. For Turbo Pascal for Windows, `flname` is of type `PChar`.

`ibwrtf` automatically opens the file `flname`. 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.

After termination, `ibcntl` is the number of bytes written. `ibcnt` is the 16-bit representation of the number of bytes written.

Device IBWRTF Function

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

When the `ibwrtf` function returns, `ibsta` holds the latest device status, `ibcntl` is the number of instruction bytes written, `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.

IBWRTF (continued) IBWRTF

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 instruction bytes are sent.

Device Function Example:

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

MS-DOS Pascal

```
var flname : flbuf;
flname[1] := 'Y';
flname[2] := '.';
flname[3] := 'D';
flname[4] := 'A';
flname[5] := 'T';
(* flname has been blank-filled.                    *)
ibwrtf (rdr,flname);
```

Turbo Pascal for Windows

```
var flname : PChar;
flname := 'Y.DAT';
ibwrtf (rdr,flname);
```

IBWRTF (continued) IBWRTF

Board Function Examples:

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`.

   IBM/MS Pascal

```
var    cmd : cbuf;
    flname : flbuf;
(* Perform addressing in preparation for           *)
(* board write.                                    *)
cmd[1] := chr(UNL);
cmd[2] := chr(16#40);           (* MTA             *)
cmd[3] := chr(16#2C);           (* LAD             *)
ibcmd (brd0,cmd,3);
flname[1] := 'Y';
flname[2] := '.';
flname[3] := 'D';
flname[4] := 'A';
flname[5] := 'T';
(* flname has been blank-filled.                   *)
ibwrtf (brd0,flname);
(* Unaddress the Talker and Listener.              *)
cmd[1] := chr(UNT);
cmd[2] := chr(UNL);
ibcmd (brd0,cmd,2);
```

IBWRTF (continued) IBWRTF

QuickPascal/Turbo Pascal

```
var    cmd : cbuf;
    flname : flbuf;
(* Perform addressing in preparation for           *)
(* board write.                                    *)
cmd[1] := chr(UNL);
cmd[2] := chr($40);              (* MTA            *)
cmd[3] := chr($2C);              (* LAD            *)
ibcmd (brd0,cmd,3);
flname[1] := 'Y';
flname[2] := '.';
flname[3] := 'D';
flname[4] := 'A';
flname[5] := 'T';
(* flname has been blank-filled.                   *)
ibwrtf (brd0,flname);
(* Unaddress the Talker and Listener.              *)
cmd[1] := chr(UNT);
cmd[2] := chr(UNL);
ibcmd (brd0,cmd,2);
```

Turbo Pascal for Windows

```
var    cmd,flname : PChar;
(* Perform addressing in preparation              *)
(* for board write.                               *)
cmd := '_'#40#2C;                (* MTA LAD        *)
ibcmd (brd0,cmd^,3);
flname := 'Y.DAT';
ibwrtf (brd0,flname);
(* Unaddress the Talker and Listener.             *)
cmd := '_?';
ibcmd (brd0,cmd^,2);
```

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: `ibwrti (ud,iarr,cnt)`

`ibwrti` is not available in Turbo Pascal and Turbo Pascal for Windows because the `ibwrt` call is adequate for sending data from any type of buffer. Pascal and QuickPascal, which have more rigid typing rules, require different procedure declarations for array buffers of different types.

`ud` represents a device or an interface board. `iarr` is the array of data to be sent. `iarr` is of type `ibuf` as defined in the header files `DECL.PAS` and `QPDECL.PAS`. `cnt` specifies the maximum number of bytes to be written. `cnt` is of type `integer4` in Microsoft Pascal, `longINT` in QuickPascal, and `integer` in IBM Pascal.

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

Device Function Examples:

1. Write 10 instruction bytes from the integer array `wrt` to the device `dvm`.

```
var wrt : ibuf;
wrt[1] := 1;
wrt[2] := 2;
wrt[3] := 3;
wrt[4] := 4;
wrt[5] := 5;
ibwrti (dvm,wrt,10);
```

IBWRTI (continued) IBWRTI

2. Write instruction bytes from integer array `wrt` terminated by a carriage return and a linefeed (hex 0A) to device `ptr`. Linefeed is the EOS character of the device.

IBM/MS Pascal

```
var wrt : ibuf;
wrt[1] := 1;
wrt[2] := 2;
wrt[3] := 3 + 16#0D * 256;
wrt[4] := 16#0A;
ibwrti (ptr,wrt,7);
```

QuickPascal

```
var wrt : ibuf;
wrt[1] := 1;
wrt[2] := 2;
wrt[3] := 3 + $0D * 256;
wrt[4] := $0A;
ibwrti (ptr,wrt,7);
```

Board Function Examples:

1. Write 10 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 @.)

```
var cmd : cbuf;
    wrt : ibuf;
cmd[1] := '?';                  (* UNL                  *)
cmd[2] := '@';                  (* MTA                  *)
cmd[3] := '/';                  (* LAD                  *)
ibcmd (brd,cmd,3);
(* Perform the board write.                            *)
wrt[1] := 1;
wrt[2] := 2;
wrt[3] := 3;
wrt[4] := 4;
wrt[5] := 5;
ibwrti (brd,wrt,10);
```

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: `ibwrtia (ud,iarr,cnt)`

`ibwrtia` is not available in Turbo Pascal and Turbo Pascal for Windows because the `ibwrta` call is adequate for sending data from any type of buffer. Pascal and QuickPascal, which have more rigid typing rules, require different procedure declarations for array buffers of different types.

`ud` represents a device or an interface board. `iarr` is the array of data to be sent. `iarr` is of type `ibuf` as defined in the header files `DECL.PAS` and `QPDECL.PAS`. `cnt` specifies the maximum number of bytes to be written. The data is sent in low-byte, high-byte order. `cnt` is of type `integer4` in Microsoft Pascal, `longINT` in QuickPascal, and `integer` in IBM Pascal.

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

Device Function Example:

Write 10 instruction bytes from integer array `wrt` to the device `dvm` while performing other processing.

IBM/MS Pascal

```
var   wrt : ibuf;
     mask : word;
wrt[1] := 1;
wrt[2] := 2;
wrt[3] := 3;
wrt[4] := 4;
wrt[5] := 5;
ibwrtia (dvm,wrt,10);
(* Perform other processing here.  Then wait       *)
(* for I/O completion or timeout.                  *)
mask := 16#4100;              (* TIMO CMPL         *)
ibwait (dvm,mask);
(* ibsta shows how the write terminated:  on       *)
(* CMPL, END, TIMO, or ERR.                        *)
if (ibsta and ERR) <> 0 then error;
```

IBWRTIA (continued) IBWRTIA

QuickPascal

```
var   wrt : ibuf;
     mask : word;
wrt[1] := 1;
wrt[2] := 2;
wrt[3] := 3;
wrt[4] := 4;
wrt[5] := 5;
ibwrtia (dvm,wrt,10);
(* Perform other processing here.  Then wait       *)
(* for I/O completion or timeout.                  *)
mask := $4100;                (* TIMO CMPL         *)
ibwait (dvm,mask);
(* ibsta shows how the write terminated:  on       *)
(* CMPL, END, TIMO, or ERR.                        *)
if (ibsta and ERR) <> 0 then error;
```

Board Function Example:

Write 10 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 @.)

IBM/MS Pascal

```
var  cmd : cbuf;
     wrt : ibuf;
    mask : word;
(* Perform addressing in preparation for boardwrite. *)
cmd[1] := '?';                (* UNL               *)
cmd[2] := '@';                (* MTA               *)
cmd[3] := '/';                (* LAD               *)
ibcmd (brd,cmd,3);
wrt[1] := 1;          (* Perform the board write.   *)
wrt[2] := 2;
wrt[3] := 3;
wrt[4] := 4;
wrt[5] := 5;
ibwrtia (brd,wrt,10);
(* Perform other processing here.  Then wait       *)
(* for I/O completion or timeout.                  *)
mask := 16#4100;              (* TIMO CMPL         *)
ibwait (bd,mask);
```

IBWRTIA (continued) IBWRTIA

QuickPascal

```
var  cmd : cbuf;
     wrt : ibuf;
    mask : word;
(* Perform addressing in preparation for           *)
(* board write.                                    *)
cmd[1] := '?';                (* UNL               *)
cmd[2] := '@';                (* MTA               *)
cmd[3] := '/';                (* LAD               *)
ibcmd (brd,cmd,3);
(* Perform the board write.                        *)
wrt[1] := 1;
wrt[2] := 2;
wrt[3] := 3;
wrt[4] := 4;
wrt[5] := 5;
ibwrtia (brd,wrt,10);
(* Perform other processing here.  Then wait       *)
(* for I/O completion or timeout.                  *)
mask := $4100;                (* TIMO CMPL         *)
ibwait (bd,mask);
```

GPIB Example Programs

These examples illustrate the programming steps that you can follow to program a representative IEEE-488 instrument from your personal computer using the NI-488 functions. The applications are written in IBM/MS Pascal, QuickPascal, Turbo Pascal, and Turbo Pascal for Windows. 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 following steps explain how to use the driver to execute certain programming and control sequences without explaining 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.

Note: In the example programs, the term *Software Reference Manual* is used to refer to the *NI-488.2 Software Reference Manual for MS-DOS.*

IBM/MS Pascal Program – Device Functions

```
PROGRAM DPSAMP(input,output);

{$INCLUDE: 'decl.pas'}

type
    string40 = lstring(40);

var
      devname : nbuf;       (* Device name buffer.  nbuf is  *)
                            (* defined in DECL.PAS as a      *)
                            (* character array.              *)
      wrt     : cbuf;       (* Data written to the Fluke 45. *)
      rd      : cbuf;       (* Data received from the        *)
                            (* Fluke 45.                     *)
      buffer  : string40;   (* Assigned the value of rd.     *)
                            (* Will be converted to          *)
                            (* numeric.                      *)
      sendstr : string40;   (* GPIB command string.          *)
      mask    : word;       (* Wait mask.                    *)
      dvm     : integer;    (* Device number.                *)
      i,m     : integer;    (* FOR loop index.               *)
      spr     : integer;    (* Serial poll response byte.    *)
      num     : real4;      (* Numeric conversion of rd.     *)
      sum     : real4;      (* Accumulator of measurements.  *)

(* ========================================================
 *                         Procedure dvmerr
 *  The dmverr procedure notifies you that the Fluke 45
 *  returned an invalid serial poll response byte.  The
 *  error message is printed along with the serial poll
 *  response byte.
 *
 *  The NI-488 function ibonl is called to disable the
 *  hardware and software.
 * ========================================================*)
procedure dvmerr(msg:string40; spr:integer);

begin

    writeln (msg);
    writeln('Status Byte = ', spr);

(*  Call the ibonl function to disable the hardware and     *)
(*  software.                                               *)

    ibonl (dvm,0);

end;
```

```
(* ============================================================
 *                        Procedure gpiberr
 *  The gpiberr procedure notifies you that an NI-488 routine
 *  failed by printing an error message.  The status variable
 *  ibsta is printed in decimal along with the mnemonic
 *  meaning of the bit position.   The status variable iberr
 *  is printed in decimal along with the mnemonic meaning
 *  of the decimal value.  The status variable ibcnt is
 *  printed in decimal.
 *
 *  The NI-488 function ibonl is called to disable the
 *  hardware and software.
 * ============================================================*)
procedure gpiberr(msg:string40);

begin

   writeln (msg);

   write('ibsta = ', ibsta,' <');
   if ibsta and ERR    <>  0 then write (' ERR');
   if ibsta and TIMO   <>  0 then write (' TIMO');
   if ibsta and EEND   <>  0 then write (' END');
   if ibsta and SRQI   <>  0 then write (' SRQI');
   if ibsta and RQS    <>  0 then write (' RQS');
   if ibsta and CMPL   <>  0 then write (' CMPL');
   if ibsta and LOK    <>  0 then write (' LOK');
   if ibsta and REM    <>  0 then write (' REM');
   if ibsta and CIC    <>  0 then write (' CIC');
   if ibsta and ATN    <>  0 then write (' ATN');
   if ibsta and TACS   <>  0 then write (' TACS');
   if ibsta and LACS   <>  0 then write (' LACS');
   if ibsta and DTAS   <>  0 then write (' DTAS');
   if ibsta and DCAS   <>  0 then write (' DCAS');
   writeln(' >');

   write('iberr = ', iberr);
   if iberr = EDVR then writeln (' EDVR <DOS Error>');
   if iberr = ECIC then writeln (' ECIC <Not CIC>');
   if iberr = ENOL then writeln (' ENOL <No Listener>');
   if iberr = EADR then writeln (' EADR <Address error>');
   if iberr = EARG then writeln (' EARG <Invalid argument>');
   if iberr = ESAC then writeln (' ESAC <Not Sys Ctrlr>');
   if iberr = EABO then writeln (' EABO <Op. aborted>');
   if iberr = ENEB then writeln (' ENEB <No GPIB board>');
   if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
   if iberr = ECAP then writeln (' ECAP <No capability>');
   if iberr = EFSO then writeln (' EFSO <File sys. error>');
   if iberr = EBUS then writeln (' EBUS <Command error>');
   if iberr = ESTB then writeln (' ESTB <Status byte lost>');
   if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
   if iberr = ETAB then writeln (' ETAB <Table Overflow>');

   writeln('ibcnt = ', ibcnt);
```

```
(*  Call the ibonl function to disable the hardware          *)
(*  and software.                                            *)

    ibonl (dvm,0);

end;

(* ========================================================
 *                      Procedure readdata
 *  The readdata procedure reads 10 measurements from the
 *  Fluke 45 and calculates the average of the measurements.
 *
 *  The return statement terminates this procedure.
 * ========================================================*)
procedure readdata;

begin

(*  Initialize the accumulator of the 10 measurements        *)
(*  to zero.                                                 *)

    sum := 0.0;

(*  Establish a FOR loop to read the 10 measurements.        *)
(*  The variable m serves as the counter of the FOR loop.    *)

   for m := 1 to 10 do
    begin

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

         ibtrg(dvm);
         if (ibsta and ERR) <> 0 then
           begin
             gpiberr('Ibtrg Error');
             return
           end;

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

         sendstr := 'VAL1?';
         for i:= 1 to 5 do
           wrt[i] := sendstr[i];

         ibwrt (dvm,wrt,5);
         if (ibsta and ERR) <> 0 then
           begin
             gpiberr('Ibwrt Error');
             return
           end;
```

```
(*  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 bitposition 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.          *)

     mask := 16#4800;                  (* RQS + TIMO *)
      ibwait(dvm, mask);
      if (ibsta and (ERR or TIMO)) <> 0 then
        begin
          gpiberr('Ibwait Error');
          return
        end;

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

      spr := 0;
      ibrsp(dvm,spr);
      if (ibsta and ERR) <> 0 then
        begin
          gpiberr('Ibrsp Error');
          return
        end;

(*  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 <> 16#50) then
        begin
          dvmerr('Fluke 45 Error', spr);
          return
        end;

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

      ibrd (dvm, rd, 10);
      if (ibsta and ERR) <> 0 then
        begin
          gpiberr('Ibrd Error');
          return
        end;

(*  Assign the array rd to the string buffer. Remove       *)
(*  spaces in buffer.  Print the measurement.              *)
```

```
          buffer.len := lobyte(ibcnt) - 1;
          for i := 1 to (ibcnt - 1) do
             buffer[i] := rd[i];
          writeln('Reading: ', buffer);
          writeln;

    (*  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     *)
    (*  the conversion, add the value to the accumulator.    *)

          if decode(buffer,num) then
            sum := sum + num
          else
            return;

     end;  (*  Continue the FOR loop until 10               *)
           (*  measurements are read.                       *)

(*  Print the average of the 10 readings.                    *)

    writeln('The average of the 10 readings is ', sum/10);

end;

(* ========================================================
 *                            MAIN
 * ========================================================*)

BEGIN

    writeln('Read 10 measurements from the Fluke 45...');
    writeln;

(*  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.   *)

    devname := 'DVM    ';
    dvm := ibfind (devname);
    if (dvm < 0) then
      begin
        gpiberr('Ibfind Error');
        return
      end;

(*  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.                                   *)
```

```
    ibclr (dvm);
    if (ibsta and ERR ) <> 0 then
      begin
        gpiberr('Ibclr Error');
        return
      end;

(*  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.  *)

    sendstr := '*RST; VAC; AUTO; TRIGGER 2; *SRE 16';
    for i:= 1 to 35 do
       wrt[i] := sendstr[i];

    ibwrt (dvm,wrt,35);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Ibwrt Error');
        return
      end;

(*  Call readdata to read 10 measurements from Fluke 45.      *)

    readdata;

(*  Call the ibonl function to disable the device DVM.        *)

    ibonl (dvm,0);

end.
```

IBM/MS Pascal Program – Board Functions

```
PROGRAM BPSAMP(input,output);
{$INCLUDE: 'decl.pas'}

type
    string40 = lstring(40);

var
    bdname    : nbuf;       (* Board name buffer.  nbuf is    *)
                            (* defined in DECL.PAS as a       *)
                            (* character array.               *)
    cmd       : cbuf;       (* Array of commands.  cbuf is    *)
                            (* defined in DECL.PAS as a       *)
                            (* character array.               *)
    wrt       : cbuf;       (* Data written to the            *)
                            (* Fluke 45.                      *)
    rd        : cbuf;       (* Data received from the         *)
                            (* Fluke 45.                      *)
    buffer    : string40;   (* Assigned the value of rd.      *)
                            (* Will be converted to numeric.  *)
    sendstr   : string40;   (* GPIB command string.           *)
    mask      : word;       (* Wait mask.                     *)
    bd        : integer;    (* Board number.                  *)
    i,m       : integer;    (* FOR loop index.                *)
    num       : real4;      (* Numeric conversion of rd.      *)
    sum       : real4;      (* Accumulator of measurements.   *)

 (* ========================================================
  *                        Procedure dvmerr
  *  The dmverr procedure notifies you that the Fluke 45
  *  returned an invalid serial poll response byte.  The
  *  error message is printed along with the serial poll
  *  response byte.
  *
  *  The NI-488 function ibonl is called to disable the
  *  hardware and software.
  *
  * ========================================================*)
procedure dvmerr(msg:string40; rdchar:char);

begin

    writeln (msg);
    writeln('Status Byte = ', ord(rdchar));

(*  Call the ibonl function to disable the hardware         *)
(*  and software.                                           *)

    ibonl (bd,0);

end;
```

```
(* ============================================================
 *                       Procedure gpiberr
 *  The gpiberr procedure notifies you that an NI-488 routine
 *  failed by printing an error message.  The status variable
 *  ibsta is printed in decimal along with the mnemonic
 *  meaning of the bit position.   The status variable iberr
 *  is printed in decimal along with the mnemonic meaning of
 *  the decimal value.  The status variable ibcnt is printed
 *  in decimal.
 *
 *  The NI-488 function ibonl is called to disable the
 *  hardware and software.
 * ============================================================*)
procedure gpiberr(msg:string40);

begin

   writeln (msg);

   write('ibsta = ', ibsta,' <');
   if ibsta and ERR    <>  0 then write (' ERR');
   if ibsta and TIMO   <>  0 then write (' TIMO');
   if ibsta and EEND   <>  0 then write (' END');
   if ibsta and SRQI   <>  0 then write (' SRQI');
   if ibsta and RQS    <>  0 then write (' RQS');
   if ibsta and CMPL   <>  0 then write (' CMPL');
   if ibsta and LOK    <>  0 then write (' LOK');
   if ibsta and REM    <>  0 then write (' REM');
   if ibsta and CIC    <>  0 then write (' CIC');
   if ibsta and ATN    <>  0 then write (' ATN');
   if ibsta and TACS   <>  0 then write (' TACS');
   if ibsta and LACS   <>  0 then write (' LACS');
   if ibsta and DTAS   <>  0 then write (' DTAS');
   if ibsta and DCAS   <>  0 then write (' DCAS');
   writeln(' >');

     write('iberr = ', iberr);
   if iberr = EDVR then writeln (' EDVR <DOS Error>');
   if iberr = ECIC then writeln (' ECIC <Not CIC>');
   if iberr = ENOL then writeln (' ENOL <No Listener>');
   if iberr = EADR then writeln (' EADR <Address error>');
   if iberr = EARG then writeln (' EARG <Invalid argument>');
   if iberr = ESAC then writeln (' ESAC <Not Sys Ctrlr>');
   if iberr = EABO then writeln (' EABO <Op. aborted>');
   if iberr = ENEB then writeln (' ENEB <No GPIB board>');
   if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
   if iberr = ECAP then writeln (' ECAP <No capability>');
   if iberr = EFSO then writeln (' EFSO <File sys. error>');
   if iberr = EBUS then writeln (' EBUS <Command error>');
   if iberr = ESTB then writeln (' ESTB <Status byte lost>');
   if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
   if iberr = ETAB then writeln (' ETAB <Table Overflow>');

   writeln('ibcnt = ', ibcnt);
```

```
(*  Call the ibonl function to disable the hardware         *)
(*  and software.                                            *)

    ibonl (bd,0);

end;

(* ========================================================
 *                       Procedure readdata
 *  The readdata procedure reads 10 measurements from the
 *  Fluke 45 and calculates the average of the measurements.
 *
 *  The return statement terminates this procedure.
 * ========================================================*)
procedure readdata;

begin

(*  Initialize the accumulator of the 10 measurements        *)
(*  to zero.                                                 *)

    sum := 0.0;

(*  Establish a FOR loop to read the 10 measurements.        *)
(*  The variable m serves as the counter of the FOR loop.    *)

    for m := 1 to 10 do
     begin

      (*  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.                    *)

          cmd[1] := '!';
          cmd[2] := '@';
          ibcmd (bd,cmd,2);
          if (ibsta and ERR) <> 0 then
            begin
              gpiberr('Ibcmd Error');
              return
            end;

       (*  Trigger the Fluke 45 by sending the trigger           *)
       (*  (GET) command (hex 08) message.  If the error         *)
       (*  bit (ERR) is set in ibsta, call gpiberr with          *)
       (*  an error message.                                     *)
```

```
    cmd[1] := chr(GET);
    ibcmd (bd,cmd,1);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Ibcmd Error');
        return
      end;

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

    sendstr := 'VAL1?';
    for i := 1 to 5 do
      wrt[i] := sendstr[i];
    ibwrt (bd,wrt,5);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Ibwrt Error');
        return
      end;

(*  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.                *)

    mask := 16#5000;           (* SRQI + TIMO *)
    ibwait(bd, mask);
    if (ibsta and (ERR or TIMO)) <> 0 then
      begin
        gpiberr('Ibwait Error');
        return
      end;

(*  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.                     *)
```

```
      cmd[1] := '_';
      cmd[2] := '?';
      cmd[3] := chr(SPE);
      cmd[4] := 'A';
      cmd[5] := ' ';
      ibcmd(bd, cmd, 5);
      if (ibsta and ERR) <> 0 then
        begin
          gpiberr('Ibcmd Error');
          return
        end;

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

      ibrd(bd, rd, 1);
      if (ibsta and ERR) <> 0 then
        begin
          gpiberr('Ibrd Error');
          return
        end;

(*  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[1]) <> 16#50) then
        begin
          dvmerr('Fluke 45 Error', rd[1]);
          return
        end;

(*  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.                   *)

      cmd[1] := chr(SPD);
      ibcmd(bd, cmd, 1);
      if (ibsta and ERR) <> 0 then
        begin
          gpiberr('Ibcmd Error');
          return
        end;
```

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

          ibrd (bd,rd,10);
          if (ibsta and ERR) <> 0 then
            begin
              gpiberr('Ibrd Error');
              return
            end;

    (*  Assign the array rd to the string buffer.  Remove     *)
    (*  spaces in buffer.  Print the measurement.             *)

          buffer.len := lobyte(ibcnt) - 1;
          for i := 1 to (ibcnt - 1) do
             buffer[i] := rd[i];
          writeln('Reading: ', buffer);
          writeln;

    (*  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      *)
    (*  the conversion, add the value to the accumulator.     *)

          if decode(buffer,num) then
            sum := sum + num
          else
            return;

     end;  (*  Continue the FOR loop until 10               *)
           (*  measurements are read.                       *)

    (*  Print the average of the 10 readings.                *)

    writeln('The average of the 10 readings is ', sum/10);

end;

(* ========================================================
 *                              MAIN
 * ========================================================*)

BEGIN

    writeln('Read 10 measurements from the Fluke 45...');
    writeln;

(*  Assign a unique identifier to board 0 and store in       *)
(*  the variable bd.  The name 'GPIB0' is the default        *)
(*  name of board 0.  If bd is less than zero, call          *)
(*  gpiberr with an error message.                           *)
```

```
    bdname := 'GPIB0  ';
    bd := ibfind (bdname);
    if (bd < 0) then
      begin
        gpiberr('Ibfind Error');
        return
      end;

(*  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.                                           *)

    ibsic (bd);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Ibsic Error');
        return
      end;

(*  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.               *)

    ibsre (bd,1);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Ibsre Error');
        return
      end;

(*  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.                           *)

    cmd[1] := chr(LLO);
    cmd[2] := chr(DCL);
    cmd[3] := '!';
    cmd[4] := '@';
    ibcmd (bd,cmd,4);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Ibcmd Error');
        return
      end;
```

```
(*  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.                                                  *)

    sendstr := '*RST; VAC; AUTO; TRIGGER 2; *SRE 16';
    for i:= 1 to 35 do
       wrt[i] := sendstr[i];

    ibwrt (bd,wrt,35);
    if (ibsta and ERR) <> 0 then
      begin
        gpiberr('Ibwrt Error');
        return
      end;

(*  Call readdata to read 10 measurements from the           *)
(*  Fluke 45.                                                *)

    readdata;

(*  Call the ibonl function to disable the hardware          *)
(*  and software.                                            *)

    ibonl (bd,0);

END.
```

QuickPascal Program – Device Functions

```
PROGRAM DQPSAMP(input,output);


{$I qpdecl.pas}

const
    maxlen = 10;              (* Maximum length of data array.  *)

var
  devname   : nbuf;           (* Device name buffer. nbuf is    *)
                              (* defined in QPDECL.PAS as a     *)
                              (* character array.               *)
  wrt       : cbuf;           (* Data written to the            *)
                              (* Fluke 45.                      *)
  rd        : cbuf;           (* Data received from the         *)
                              (* Fluke 45.                      *)
  buffer    : string[maxlen];(* Assigned the value of rd.      *)
                              (* Will be converted to           *)
                              (* numeric.                       *)
  sendstr   : string[40];     (* GPIB command string.           *)
  mask      : integer;        (* Wait mask.                     *)
  dvm       : integer;        (* Device number.                 *)
  i,m       : integer;        (* FOR loop index.                *)
  spr       : integer;        (* Serial poll response byte.     *)
  code      : integer;        (* Procedure VAL parameter.       *)
                              (* VAL is Turbo Pascal            *)
                              (* conversion procedure.          *)
  num       : real;           (* Numeric conversion of rd.      *)
  sum       : real;           (* Accumulator of measurements.   *)

(* ========================================================
 *                        Procedure dvmerr
 *  The dvmerr procedure notifies you that the Fluke 45
 *  returned an invalid serial poll response byte.  The
 *  error message is printed along with the serial poll
 *  response byte.

 *  The NI-488 function ibonl is called to disable the
 *  hardware and software.
 *
 *  The halt procedure terminates this program.
 * ========================================================*)
procedure dvmerr(msg:string; spr:integer);

begin

    writeln (msg);
    writeln('Status Byte = ', spr);

(*  Call the ibonl function to disable the hardware         *)
(*  and software.                                           *)
```

```
    ibonl (dvm,0);

    halt;

end;

(* ============================================================
 *                        Procedure gpiberr
 *  The gpiberr procedure notifies you that an NI-488 routine
 *  failed by printing an error message.  The status variable
 *  ibsta is printed in decimal along with the mnemonic
 *  meaning of the bit position.  The status variable iberr
 *  is printed in decimal along with the mnemonic meaning of
 *  the decimal value.  The status variable ibcnt is printed
 *  in decimal.
 *
 *  The NI-488 function ibonl is called to disable the
 *  hardware and software.
 *
 *  The halt command stops execution of this program.
 * ============================================================*)
procedure gpiberr(msg:string);

begin

   writeln (msg);

   write('ibsta = ', ibsta,' <');
   if ibsta and ERR    <>  0 then write (' ERR');
   if ibsta and TIMO   <>  0 then write (' TIMO');
   if ibsta and EEND   <>  0 then write (' END');
   if ibsta and SRQI   <>  0 then write (' SRQI');
   if ibsta and RQS    <>  0 then write (' RQS');
   if ibsta and CMPL   <>  0 then write (' CMPL');
   if ibsta and LOK    <>  0 then write (' LOK');
   if ibsta and REM    <>  0 then write (' REM');
   if ibsta and CIC    <>  0 then write (' CIC');
   if ibsta and ATN    <>  0 then write (' ATN');
   if ibsta and TACS   <>  0 then write (' TACS');
   if ibsta and LACS   <>  0 then write (' LACS');
   if ibsta and DTAS   <>  0 then write (' DTAS');
   if ibsta and DCAS   <>  0 then write (' DCAS');
   writeln(' >');

   write('iberr = ', iberr);
   if iberr = EDVR then writeln (' EDVR <DOS Error>');
   if iberr = ECIC then writeln (' ECIC <Not CIC>');
   if iberr = ENOL then writeln (' ENOL <No Listener>');
   if iberr = EADR then writeln (' EADR <Address error>');
   if iberr = EARG then writeln (' EARG <Invalid argument>');
   if iberr = ESAC then writeln (' ESAC <Not Sys Ctrlr>');
   if iberr = EABO then writeln (' EABO <Op. aborted>');
   if iberr = ENEB then writeln (' ENEB <No GPIB board>');
   if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
```

```
   if iberr = ECAP then writeln (' ECAP <No capability>');
   if iberr = EFSO then writeln (' EFSO <File sys. error>');
   if iberr = EBUS then writeln (' EBUS <Command error>');
   if iberr = ESTB then writeln (' ESTB <Status byte lost>');
   if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
   if iberr = ETAB then writeln (' ETAB <Table Overflow>');

   writeln('ibcnt = ', ibcnt);

(*  Call the ibonl function to disable the hardware          *)
(*  and software.                                            *)

    ibonl (dvm,0);

    halt;

end;

(* ========================================================
 *                       Procedure readdata
 *  The readdata procedure reads 10 measurements from the
 *  Fluke 45 and calculates the average of the measurements.
 *
 *  The exit procedure terminates this procedure.
 * ========================================================*)
procedure readdata;

begin

(*  Initialize the accumulator of the 10 measurements        *)
(*  to zero.                                                 *)

    sum := 0.0;

(*  Establish a FOR loop to read the 10 measurements.        *)
(*  The variable m serves as the counter of the FOR loop.    *)

    for m := 1 to 10 do
     begin

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

         ibtrg(dvm);
         if (ibsta and ERR) <> 0 then gpiberr('Ibtrg Error');

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

         sendstr := 'VAL1?';
         for i:= 1 to 5 do
           wrt[i] := sendstr[i];
```

```
        ibwrt (dvm,wrt,5);
        if (ibsta and ERR) <> 0 then gpiberr('Ibwrt Error');

   (*    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 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.          *)

        mask := $4800;                (* RQS + TIMO *)
        ibwait(dvm, mask);
        if (ibsta and (ERR or TIMO)) <> 0 then
           gpiberr('Ibwait Error');

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

        spr := 0;
        ibrsp(dvm,spr);
        if (ibsta and ERR) <> 0 then gpiberr('Ibrsp Error');

  (*  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 <> $50) then dvmerr('Fluke 45 Error', spr);

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

        ibrd (dvm, rd, 10);
        if (ibsta and ERR) <> 0 then gpiberr('Ibrd Error');

  (*  Assign the array rd to the string buffer.  Remove      *)
  (*  spaces in buffer.  Print the measurement.              *)

        for i:= 1 to (ibcnt - 1) do
           buffer[i] := rd[i];
        buffer[0] := chr(ibcnt - 1);
        delete(buffer, ibcnt, maxlen - ibcnt + 1);
        writeln('Reading: ', buffer);
        writeln;

  (*  Convert the measurement to its numeric value.  If     *)
  (*  there is an error during the conversion, print the    *)
  (*  index of the character that did not convert and       *)
  (*  terminate this program. If an error does not occur    *)
  (*  during the conversion, add the value to the           *)
  (*  accumulator.
```

```
          val(buffer, num, code);
          if code <> 0 then
            begin
              writeln('Error at position: ', code);
              exit;
            end
          else
            sum := sum + num;

     end; (*  Continue the FOR loop until 10 measurements   *)
          (*  are read.                                     *)

     (*  Print the average of the 10 readings.             *)

     writeln('The average of the 10 readings is ', sum/10);

end;

(* ============================================================
 *                            MAIN
 * ============================================================*)

BEGIN

    writeln('Read 10 measurements from the Fluke 45...');
    writeln;

(*  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.  *)

    devname := 'DVM    ';
    dvm := ibfind (devname);
    if (dvm < 0) then gpiberr('Ibfind Error');

(*  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.                                  *)

    ibclr (dvm);
    if (ibsta and ERR ) <> 0 then gpiberr('Ibclr Error');

(*  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.           *)
```

```
    sendstr := '*RST; VAC; AUTO; TRIGGER 2; *SRE 16';
    for i:= 1 to 35 do
       wrt[i] := sendstr[i];

    ibwrt (dvm,wrt,35);
    if (ibsta and ERR) <> 0 then gpiberr('Ibwrt Error');

(*  Call readdata to read 10 measurements from the          *)
(*  Fluke 45.                                               *)

    readdata;

(*  Call the ibonl function to disable the device DVM.      *)

    ibonl (dvm,0);

end.
```

QuickPascal Program – Board Functions

```
PROGRAM BQPSAMP(input,output);

{$I qpdecl.pas}

const
  maxlen = 10;               (* Maximum length of data       *)
                             (* array.                       *)

Var
  bdname    : nbuf;          (* Board name buffer.  nbuf     *)
                             (* is defined in QPDECL.PAS as  *)
                             (* a character array.           *)
  cmd       : cbuf;          (* Array of commands. cbuf is   *)
                             (* defined in QPECL.PAS as a    *)
                             (* character array.             *)
  wrt       : cbuf;          (* Data written to the Fluke 45.*)
  rd        : cbuf;          (* Data received from the       *)
                             (* Fluke 45.                    *)
  buffer    : string[maxlen];(* Assigned the value of rd.    *)
                             (* Will be converted to         *)
                             (* numeric.                     *)
  sendstr   : string[40];    (* GPIB command string.         *)
  mask      : integer;       (* Wait mask.                   *)
  bd        : integer;       (* Board number.                *)
  i,m       : integer;       (* FOR loop index.              *)
  code      : integer;       (* Procedure VAL parameter.     *)
                             (* VAL is a Turbo Pascal        *)
                             (* conversion procedure.        *)
  num       : real;          (* Numeric conversion of rd.    *)
  sum       : real;          (* Accumulator of measurements. *)

(* ========================================================
 *                        Procedure dvmerr
 *  The dvmerr procedure notifies you that the Fluke 45
 *  returned an invalid serial poll response byte.  The
 *  error message is printed along with the serial poll
 *  response byte.
 *
 *  The NI-488 function ibonl is called to disable the
 *  hardware and software.
 *
 *  The halt procedure terminates this procedure.
 * ========================================================*)
procedure dvmerr(msg:string; rdchar:char);

begin

    writeln (msg);
    writeln('Status Byte = ', ord(rdchar));
```

```
(*  Call the ibonl function to disable the hardware         *)
(*  and software.                                            *)

    ibonl (bd,0);

    halt;

end;

(* ============================================================
 *                          Procedure gpiberr
 *  The gpiberr procedure notifies you that an NI-488 routine
 *  failed by printing an error message.  The status variable
 *  ibsta is printed in decimal along with the mnemonic
 *  meaning of the bit position.  The status variable iberr
 *  is 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.
 *
 *  The halt procedure stops execution of this program.
 * ============================================================*)
 procedure gpiberr(msg:string);

 begin

   writeln (msg);

   write('ibsta = ', ibsta,' <');
   if ibsta and ERR    <>  0 then write (' ERR');
   if ibsta and TIMO   <>  0 then write (' TIMO');
   if ibsta and EEND   <>  0 then write (' END');
   if ibsta and SRQI   <>  0 then write (' SRQI');
   if ibsta and RQS    <>  0 then write (' RQS');
   if ibsta and CMPL   <>  0 then write (' CMPL');
   if ibsta and LOK    <>  0 then write (' LOK');
   if ibsta and REM    <>  0 then write (' REM');
   if ibsta and CIC    <>  0 then write (' CIC');
   if ibsta and ATN    <>  0 then write (' ATN');
   if ibsta and TACS   <>  0 then write (' TACS');
   if ibsta and LACS   <>  0 then write (' LACS');
   if ibsta and DTAS   <>  0 then write (' DTAS');
   if ibsta and DCAS   <>  0 then write (' DCAS');
   writeln(' >');

   write('iberr = ', iberr);
   if iberr = EDVR then writeln (' EDVR <DOS Error>');
   if iberr = ECIC then writeln (' ECIC <Not CIC>');
   if iberr = ENOL then writeln (' ENOL <No Listener>');
   if iberr = EADR then writeln (' EADR <Address error>');
   if iberr = EARG then writeln (' EARG <Invalid argument>');
   if iberr = ESAC then writeln (' ESAC <Not Sys Ctrlr>');
   if iberr = EABO then writeln (' EABO <Op. aborted>');
   if iberr = ENEB then writeln (' ENEB <No GPIB board>');
```

```
   if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
   if iberr = ECAP then writeln (' ECAP <No capability>');
   if iberr = EFSO then writeln (' EFSO <File sys. error>');
   if iberr = EBUS then writeln (' EBUS <Command error>');
   if iberr = ESTB then writeln (' ESTB <Status byte lost>');
   if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
   if iberr = ETAB then writeln (' ETAB <Table Overflow>');

   writeln('ibcnt = ', ibcnt);

(*  Call the ibonl function to disable the hardware         *)
(*  and software.                                           *)

    ibonl (bd,0);

    halt;

end;
(* ========================================================
 *                      Procedure readdata
 *  The readdata procedure reads 10 measurements from the
 *  Fluke 45 and calculates the average of the measurements.
 *
 *  The exit procedure terminates this procedure.
 * ========================================================*)
procedure readdata;

begin

(*  Initialize the accumulator of the 10 measurements       *)
(*  to zero.                                                *)

    sum := 0.0;

(*  Establish a FOR loop to read the 10 measurements.       *)
(*  The variable m serves as the counter of the FOR loop.   *)

   for m := 1 to 10 do
    begin

   (*  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.                                 *)

        cmd[1] := '!';
        cmd[2] := '@';
        ibcmd (bd,cmd,2);
        if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');
```

```
(*  Trigger the Fluke 45 by sending the trigger (GET)    *)
(*  command (hex 08) message.  If the error bit (ERR) is    *)
(*  set in ibsta, call gpiberr with an error message.    *)

    cmd[1] := chr(GET);
    ibcmd (bd,cmd,1);
    if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');

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

      sendstr := 'VAL1?';
      for i := 1 to 5 do
        wrt[i] := sendstr[i];
      ibwrt (bd,wrt,5);
      if (ibsta and ERR) <> 0 then gpiberr('Ibwrt Error');

(*  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 isdetected 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.                              *)

    mask := $5000;             (* SRQI + TIMO *)
    ibwait(bd, mask);
    if (ibsta and (ERR or TIMO) <> 0 then
         gpiberr('Ibwait Error');

(*  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.                                      *)

    cmd[1] := '_';
    cmd[2] := '?';
    cmd[3] := chr(SPE);
    cmd[4] := 'A';
    cmd[5] := ' ';
    ibcmd(bd, cmd, 5);
    if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');
```

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

    ibrd(bd, rd, 1);
    if (ibsta and ERR) <> 0 then gpiberr('Ibrd Error');

(*  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[1]) <> $50) then dvmerr('Fluke 45 Error',
                                     rd[1]);

(*  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.      *)

    cmd[1] := chr(SPD);
    ibcmd(bd, cmd, 1);
    if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');

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

    ibrd (bd,rd,10);
    if (ibsta and ERR) <> 0 then gpiberr('Ibrd Error');

(*  Assign the array rd to the string buffer.  Remove  *)
(*  spaces in buffer.  Print the measurement.          *)

    for i:= 1 to (ibcnt - 1) do
       buffer[i] := rd[i];
    buffer[0] := chr(ibcnt - 1);
    delete(buffer, ibcnt, maxlen - ibcnt + 1);
    writeln('Reading: ', buffer);
    writeln;

(*  Convert the measurement to its numeric value.  If  *)
(*  there is an error during the conversion, print     *)
(*  the index of the character that did not convert    *)
(*  and terminate this program.  If an error does not  *)
(*  occur during the conversion, add the value to the  *)
(*  accumulator.                                       *)
```

```
          val(buffer, num, code);
          if code <> 0 then
            begin
              writeln('Error at position: ', code);
              exit;
            end
          else
            sum := sum + num;

     end;  (*  Continue the FOR loop until 10               *)
           (*  measurements are read.                       *)

(*  Print the average of the 10 readings.                    *)

    writeln('The average of the 10 readings is ', sum/10);

end;

(* ============================================================
 *                            MAIN
 * ============================================================*)

BEGIN

    writeln('Read 10 measurements from the Fluke 45...');
    writeln;

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

    bdname := 'GPIB0  ';
    bd := ibfind (bdname);
    if (bd < 0) then gpiberr('Ibfind Error');

(*  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.                                                 *)

    ibsic (bd);
    if (ibsta and ERR) <> 0 then gpiberr('Ibsic Error');

(*  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.               *)

    ibsre (bd,1);
    if (ibsta and ERR) <> 0 then gpiberr('Ibsre Error');
```

```
(*  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.            *)

    cmd[1] := chr(LLO);
    cmd[2] := chr(DCL);
    cmd[3] := '!';
    cmd[4] := '@';
    ibcmd (bd,cmd,4);
    if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');

(*  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.                                   *)

    sendstr := '*RST; VAC; AUTO; TRIGGER 2; *SRE 16';
    for i:= 1 to 35 do
       wrt[i] := sendstr[i];

    ibwrt (bd,wrt,35);
    if (ibsta and ERR) <> 0 then gpiberr('Ibwrt Error');

(*  Call readdata to read 10 measurements from the           *)
(*  Fluke 45.                                                *)

(*  Call the ibonl function to disable the hardware          *)
(*  and software.                                            *)

    ibonl (bd,0);

END.
```

Turbo Pascal Program – Device Functions

```
PROGRAM DTPSAMP(input,output);

uses tpdecl;

const
  maxlen = 10;              (* Maximum length of data array    *)

var
  devname   : nbuf;           (* Device name buffer. nbuf is  *)
                              (* defined in TPDECL.PAS as a   *)
                              (* character array.             *)
  wrt       : cbuf;           (* Data written to the          *)
                              (* Fluke 45.                    *)
  rd        : cbuf;           (* Data received from the       *)
                              (* Fluke 45.                    *)
  buffer    : string[maxlen];(* Assigned the value of rd.     *)
                              (* Will be converted to         *)
                              (* numeric.                     *)
  sendstr   : string[40];     (* GPIB command string.         *)
  mask      : integer;        (* Wait mask.                   *)
  dvm       : integer;        (* Device number.               *)
  i,m       : integer;        (* FOR loop index.              *)
  spr       : integer;        (* Serial poll response byte.   *)
  code      : integer;        (* Procedure VAL parameter.     *)
                              (* VAL is Turbo Pascal          *)
                              (* conversion procedure.        *)
  num       : real;           (* Numeric conversion of rd.    *)
  sum       : real;           (* Accumulator of measurements. *)

(* ========================================================
 *                      Procedure dvmerr
 *  The dvmerr procedure notifies you that the Fluke 45
 *  returned an invalid serial poll response byte.  The error
 *  message is printed along with the serial poll response
 *  byte.

 *  The NI-488 function ibonl is called to disable the
 *  hardware and software.

 *  The halt procedure terminates this program.
 * ========================================================*)
procedure dvmerr(msg:string; spr:integer);

begin

    writeln (msg);
    writeln('Status Byte = ', spr);

(*  Call the ibonl function to disable the hardware         *)
(*  and software.                                           *)

    ibonl (dvm,0);

    halt;

end;
```

```
(* ========================================================
 *                        Procedure gpiberr
 *  The gpiberr procedure notifies you that an NI-488 routine
 *  failed by printing an error message.  The status variable
 *  ibsta is printed in decimal along with the mnemonic
 *  meaning of the bit position.  The status variable iberr
 *  is printed in decimal along with the mnemonic meaning of
 *  the decimal value.  The status variable ibcnt is printed
 *  in decimal.
 *
 *  The NI-488 function ibonl is called to disable the
 *  hardware and software.
 *
 *  The halt procedure stops execution of this program.
 * ========================================================*)
procedure gpiberr(msg:string);

begin

   writeln (msg);

   write('ibsta = ', ibsta,' <');
   if ibsta and ERR    <>  0 then write (' ERR');
   if ibsta and TIMO   <>  0 then write (' TIMO');
   if ibsta and EEND   <>  0 then write (' END');
   if ibsta and SRQI   <>  0 then write (' SRQI');
   if ibsta and RQS    <>  0 then write (' RQS');
   if ibsta and CMPL   <>  0 then write (' CMPL');
   if ibsta and LOK    <>  0 then write (' LOK');
   if ibsta and REM    <>  0 then write (' REM');
   if ibsta and CIC    <>  0 then write (' CIC');
   if ibsta and ATN    <>  0 then write (' ATN');
   if ibsta and TACS   <>  0 then write (' TACS');
   if ibsta and LACS   <>  0 then write (' LACS');
   if ibsta and DTAS   <>  0 then write (' DTAS');
   if ibsta and DCAS   <>  0 then write (' DCAS');
   writeln(' >');

   write('iberr = ', iberr);
   if iberr = EDVR then writeln (' EDVR <DOS Error>');
   if iberr = ECIC then writeln (' ECIC <Not CIC>');
   if iberr = ENOL then writeln (' ENOL <No Listener>');
   if iberr = EADR then writeln (' EADR <Address error>');
   if iberr = EARG then writeln (' EARG <Invalid argument>');
   if iberr = ESAC then writeln (' ESAC <Not Sys Ctrlr>');
   if iberr = EABO then writeln (' EABO <Op. aborted>');
   if iberr = ENEB then writeln (' ENEB <No GPIB board>');
   if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
   if iberr = ECAP then writeln (' ECAP <No capability>');
   if iberr = EFSO then writeln (' EFSO <File sys. error>');
   if iberr = EBUS then writeln (' EBUS <Command error>');
   if iberr = ESTB then writeln (' ESTB <Status byte lost>');
   if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
   if iberr = ETAB then writeln (' ETAB <Table Overflow>');
    writeln('ibcnt = ', ibcnt);
```

```
(*  Call the ibonl function to disable the hardware         *)
(*  and software.                                            *)

     ibonl (dvm,0);

     halt;

end;

(* ========================================================
 *                      Procedure readdata
 *  The readdata procedure reads 10 measurements from the
 *  Fluke 45 and calculates the average of the measurements.
 *
 *  The exit procedure terminates this procedure.
 * ========================================================*)
procedure readdata;

begin

(*  Initialize the accumulator of the 10 measurements       *)
(*  to zero.                                                 *)

    sum := 0.0;

(*  Establish a FOR loop to read the 10 measurements.       *)
(*  The variable m serves as the counter of the FOR loop.   *)

    for m := 1 to 10 do
     begin

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

        ibtrg(dvm);
        if (ibsta and ERR) <> 0 then gpiberr('Ibtrg Error');

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

        sendstr := 'VAL1?';
        for i:= 1 to 5 do
          wrt[i] := sendstr[i];

       bwrt (dvm,wrt,5);
          if (ibsta and ERR) <> 0 then gpiberr('Ibwrt Error');

    (*  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.        *)

      mask := $4800;                    (* RQS + TIMO *)
      ibwait(dvm, mask);
      if (ibsta and (ERR or TIMO)) <> 0 then
         gpiberr('Ibwait Error');

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

      spr := 0;
      ibrsp(dvm,spr);
      if (ibsta and ERR) <> 0 then gpiberr('Ibrsp Error');

(*  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 <> $50) then dvmerr('Fluke 45 Error', spr);

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

      ibrd (dvm, rd, 10);
      if (ibsta and ERR) <> 0 then gpiberr('Ibrd Error');

(*  Assign the array rd to the string buffer.  Remove    *)
(*  spaces in buffer.  Print the measurement.            *)

    for i:= 1 to (ibcnt - 1) do
       buffer[i] := rd[i];
    buffer[0] := chr(ibcnt - 1);
    writeln('Reading: ', buffer);
    writeln;

(*  Convert the measurement to its numeric value.       *)
(*  If there is an error during the conversion, print   *)
(*  the index of the character that did not convert     *)
(*  and terminate this program.  If an error does not   *)
(*  occur during the conversion, add the value to       *)
(*  the accumulator.                                    *)

      val(buffer, num, code);
      if code <> 0 then
        begin
          writeln('Error at position: ', code);
          exit;
        end
      else
        sum := sum + num;
```

```
     end;  (*  Continue the FOR loop until 10               *)
           (*  measurements are read.                       *)

(*  Print the average of the 10 readings.                   *)

    writeln('The average of the 10 readings is ', sum/10);

end;

(* ========================================================
 *                              MAIN
 * ========================================================*)

BEGIN

    writeln('Read 10 measurements from the Fluke 45...');
    writeln;

(*  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.   *)

    devname := 'DVM    ';
    dvm := ibfind (devname);
    if (dvm < 0) then gpiberr('Ibfind Error');

(*  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.                      *)

    ibclr (dvm);
    if (ibsta and ERR ) <> 0 then gpiberr('Ibclr Error');

(*  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.                                   *)

    sendstr := '*RST; VAC; AUTO; TRIGGER 2; *SRE 16';
    for i:= 1 to 35 do
       wrt[i] := sendstr[i];

    ibwrt (dvm,wrt,35);
    if (ibsta and ERR) <> 0 then gpiberr('Ibwrt Error');

(*  Call readdata to read 10 measurements from the           *)
(*  Fluke 45.                                                *)
```

```
    readdata;
(*  Call the ibonl function to disable the device DVM.       *)
    ibonl (dvm,0);
end.
```

Turbo Pascal Program – Board Functions

```
PROGRAM BTPSAMP(input,output);

uses tpdecl;

const
  maxlen = 10;              (* Maximum length of data array.   *)

Var
  bdname  : nbuf;           (* Board name buffer.  nbuf is     *)
                            (* defined in TPDECL.PAS as a      *)
                            (* character array.                *)
  cmd     : cbuf;           (* Array of commands.  cbuf is     *)
                            (* defined in TPDECL.PAS as a      *)
                            (* character array.                *)
  wrt     : cbuf;           (* Data written to the Fluke 45.   *)
  rd      : cbuf;           (* Data received from the          *)
                            (* Fluke 45.                       *)
  buffer  : string[maxlen];(* Assigned the value of rd.        *)
                            (* Will be converted to numeric.   *)
  sendstr: string[40];      (* GPIB command string.            *)
  mask    : integer;        (* Wait mask.                      *)
  bd      : integer;        (* Board number.                   *)
  i,m     : integer;        (* FOR loop index.                 *)
  code    : integer;        (* Procedure VAL parameter.        *)
                            (* VAL is a Turbo Pascal           *)
                            (* conversion procedure.           *)
  num     : real;           (* Numeric conversion of rd.       *)
  sum     : real;           (* Accumulator of measurements.    *)

(* ========================================================
 *                          Procedure dvmerr
 *  The dvmerr procedure notifies you that the Fluke 45
 *  returned an invalid serial poll response byte.  The
 *  error message is printed along with the serial poll
 *  response byte.

 *  The NI-488 function ibonl is called to disable the
 *  hardware and software.

 *  The halt procedure terminates this program.
 * ========================================================*)
procedure dvmerr(msg:string; rdchar:char);

begin

    writeln (msg);
    writeln('Status Byte = ', ord(rdchar));

(*  Call the ibonl function to disable the hardware         *)
(*  and software.                                           *)

    ibonl (bd,0);

    halt;

end;
```

```
(* ============================================================
 *                        Procedure gpiberr
 *  The gpiberr procedure notifies you that an NI-488 routine
 *  failed by printing an error message.  The status variable
 *  ibsta is printed in decimal along with the mnemonic
 *  meaning of the bit position.  The status variable iberr
 *  is printed in decimal along with the mnemonic meaning of
 *  the decimal value.  The status variable ibcnt is printed
 *  in decimal.
 *
 *  The NI-488 function ibonl is called to disable the
 *  hardware and software.
 *
 *  The halt procedure stops execution of this program.
 * ============================================================*)
procedure gpiberr(msg:string);

begin

   writeln (msg);

   write('ibsta = ', ibsta,' <');
   if ibsta and ERR    <>  0 then write (' ERR');
   if ibsta and TIMO   <>  0 then write (' TIMO');
   if ibsta and EEND   <>  0 then write (' END');
   if ibsta and SRQI   <>  0 then write (' SRQI');
   if ibsta and RQS    <>  0 then write (' RQS');
   if ibsta and CMPL   <>  0 then write (' CMPL');
   if ibsta and LOK    <>  0 then write (' LOK');
   if ibsta and REM    <>  0 then write (' REM');
   if ibsta and CIC    <>  0 then write (' CIC');
   if ibsta and ATN    <>  0 then write (' ATN');
   if ibsta and TACS   <>  0 then write (' TACS');
   if ibsta and LACS   <>  0 then write (' LACS');
   if ibsta and DTAS   <>  0 then write (' DTAS');
   if ibsta and DCAS   <>  0 then write (' DCAS');
   writeln(' >');


   write('iberr = ', iberr);
   if iberr = EDVR then writeln (' EDVR <DOS Error>');
   if iberr = ECIC then writeln (' ECIC <Not CIC>');
   if iberr = ENOL then writeln (' ENOL <No Listener>');
   if iberr = EADR then writeln (' EADR <Address error>');
   if iberr = EARG then writeln (' EARG <Invalid argument>');
   if iberr = ESAC then writeln (' ESAC <Not Sys Ctrlr>');
   if iberr = EABO then writeln (' EABO <Op. aborted>');
   if iberr = ENEB then writeln (' ENEB <No GPIB board>');
   if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
   if iberr = ECAP then writeln (' ECAP <No capability>');
   if iberr = EFSO then writeln (' EFSO <File sys. error>');
   if iberr = EBUS then writeln (' EBUS <Command error>');
   if iberr = ESTB then writeln (' ESTB <Status byte lost>');
   if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
   if iberr = ETAB then writeln (' ETAB <Table Overflow>');
```

```
   writeln('ibcnt = ', ibcnt);

(*  Call the ibonl function to disable the hardware         *)
(*  and software.                                           *)

    ibonl (bd,0);

    halt;

end;

(* ========================================================
 *                      Procedure readdata
 *  The readdata procedure reads 10 measurements from the
 *  Fluke 45 and calculates the average of the measurements.
 *
 *  The exit procedure terminates this procedure.
 * ========================================================*)
procedure readdata;

begin

(*  Initialize the accumulator of the 10 measurements       *)
(*  to zero.                                                *)

(*  Establish a FOR loop to read the 10 measurements.       *)
(*  The variable m serves as the counter of the FOR loop.   *)

    for m := 1 to 10 do
     begin

    (*  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.       *)

          cmd[1] := '!';
          cmd[2] := '@';
          ibcmd (bd,cmd,2);
          if

    (*  Trigger the Fluke 45 by sending the trigger (GET)   *)
    (*  command (hex 08) message. If the error bit (ERR) is  *)
    (*  set in ibsta, call gpiberr with an error message.   *)

          cmd[1] := chr(GET);
          ibcmd (bd,cmd,1);
          if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');
```

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

      sendstr := 'VAL1?';
      for i := 1 to 5 do
        wrt[i] := sendstr[i];
      ibwrt (bd,wrt,5);
      if (ibsta and ERR) <> 0 then gpiberr('Ibwrt Error');

(*  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) orthe timeout bit (TIMO) is       *)
(*  set in ibsta, call gpiberr with an error message.    *)

      mask := $5000;             (* SRQI + TIMO *)
      ibwait(bd, mask);
      if (ibsta and (ERR or TIMO)) <> 0 then
        gpiberr('Ibwait Error');

(*  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.                                       *)

      cmd[1] := '_';
      cmd[2] := '?';
      cmd[3] := chr(SPE);
      cmd[4] := 'A';
      cmd[5] := ' ';
      ibcmd(bd, cmd, 5);
      if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');

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

      ibrd(bd, rd, 1);
      if (ibsta and ERR) <> 0 then gpiberr('Ibrd Error');
```

```
(*  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[1]) <> $50) then
        dvmerr('Fluke 45 Error', rd[1]);

(*  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.    *)

      cmd[1] := chr(SPD);
      ibcmd(bd, cmd, 1);
      if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');

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

      ibrd (bd,rd,10);
      if (ibsta and ERR) <> 0 then gpiberr('Ibrd Error');

(*  Assign the array rd to the string buffer.            *)
(*  Remove spaces in buffer.  Print the measurement.     *)

      for i:= 1 to (ibcnt - 1) do
         buffer[i] := rd[i];
      buffer[0] := chr(ibcnt - 1);
      writeln('Reading: ', buffer);
      writeln;

(*  Convert the measurement to its numeric value.        *)
(*  If there is an error during the conversion,          *)
(*  print the index of the character that did not        *)
(*  convert and terminate this program. If an error      *)
(*  does not occur during the conversion, add the        *)
(*  value to the accumulator.                            *)

      val(buffer, num, code);
      if code <> 0 then
        begin
          writeln('Error at position: ', code);
          exit;
        end
      else
        sum := sum + num;

 end;  (*  Continue the FOR loop until 10               *)
       (*  measurements are read.                      *)
```

```
    (*  Print the average of the 10 readings.                *)

    writeln('The average of the 10 readings is ', sum/10);

end;

(* ======================================================
 *                              MAIN
 * ======================================================*)

BEGIN

    writeln('Read 10 measurements from the Fluke 45...');
    writeln;

(*  Assign a unique identifier to board 0 and store in       *)
(*  the variable bd.  The name GPIB0 is the default name     *)
(*  of board 0.  If bd is less than zero, call gpiberr       *)
(*  with an error message.                                   *)

    bdname := 'GPIB0  ';
    bd := ibfind (bdname);
    if (bd < 0) then gpiberr('Ibfind Error');

(*  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.                                           *)

    ibsic (bd);
    if (ibsta and ERR) <> 0 then gpiberr('Ibsic Error');

(*  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.     *)

    ibsre (bd,1);
    if (ibsta and ERR) <> 0 then gpiberr('Ibsre Error');

(*  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.       *)
```

```
    cmd[1] := chr(LLO);
    cmd[2] := chr(DCL);
    cmd[3] := '!';
    cmd[4] := '@';
    ibcmd (bd,cmd,4);
    if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');

(* 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.                                                   *)

    sendstr := '*RST; VAC; AUTO; TRIGGER 2; *SRE 16';
    for i:= 1 to 35 do
       wrt[i] := sendstr[i];
    ibwrt (bd,wrt,35);
    if (ibsta and ERR) <> 0 then gpiberr('Ibwrt Error');

(*  Call readdata to read 10 measurements from the            *)
(*  Fluke 45.                                                 *)

    readdata;

(*  Call the ibonl function to disable the hardware           *)
(*  and software.                                             *)

    ibonl (bd,0);

END.
```

Turbo Pascal for Windows Program – Device Functions

```
PROGRAM DTPWSAMP(input,output);

uses wincrt, tpwgpib;

var
  wrt   : PChar;                (* Data written to the        *)
                                (* Fluke 45.                  *)
  rd    : array[0..10] of char; (* Data received from the     *)
                                (* Fluke 45.                  *)
  mask  : word;                 (* Wait mask.                 *)
  dvm   : integer;              (* Device number.             *)
  m     : integer;              (* FOR loop index.            *)
  spr   : integer;              (* Serial poll response       *)
                                (* byte.                      *)
  code  : integer;              (* Procedure VAL parameter.   *)
                                (* VAL is Turbo Pascal        *)
                                (* conversion procedure.      *)
  num   : real;                 (* Numeric conversion of RD.  *)
  sum   : real;                 (* Accumulator of             *)
                                (* measurements.              *)

(* ========================================================
 *                       Procedure DVMERR
 *  This function 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.
 *
 *  The HALT function will terminate this program.
 * ========================================================*)
procedure dvmerr(msg:string; spr:integer);

begin

    writeln (msg);
    writeln('Status Byte = ', spr);

(*  Call the IBONL function to disable the hardware and       *)
(*  software.                                                 *)

    ibonl (dvm,0);

    halt;

end;
```

```
(* ========================================================
 *                       Procedure GPIBERR
 *  This procedure will notify you that a NI-488 function
 *  failed by printing an error message. The status variable
 *  IBSTA will be printed in decimal 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 IBCNTL
 *  will be printed in decimal.
 *
 *  The NI-488 function IBONL is called to disable the
 *  hardware and software.
 *
 *  The HALT command stops execution of this program.
 * ========================================================*)
procedure gpiberr(msg:string);

begin

   writeln (msg);

   write('ibsta = ', ibsta,' <');
   if ibsta and ERR  <> 0 then write (' ERR');
   if ibsta and TIMO <> 0 then write (' TIMO');
   if ibsta and EEND <> 0 then write (' END');
   if ibsta and SRQI <> 0 then write (' SRQI');
   if ibsta and RQS  <> 0 then write (' RQS');
   if ibsta and CMPL <> 0 then write (' CMPL');
   if ibsta and LOK  <> 0 then write (' LOK');
   if ibsta and REM  <> 0 then write (' REM');
   if ibsta and CIC  <> 0 then write (' CIC');
   if ibsta and ATN  <> 0 then write (' ATN');
   if ibsta and TACS <> 0 then write (' TACS');
   if ibsta and LACS <> 0 then write (' LACS');
   if ibsta and DTAS <> 0 then write (' DTAS');
   if ibsta and DCAS <> 0 then write (' DCAS');
   writeln( ' >');

   write('iberr = ', iberr);
   if iberr = EDVR then writeln (' EDVR <DOS Error>');
   if iberr = ECIC then writeln (' ECIC <Not CIC>');
   if iberr = ENOL then writeln (' ENOL <No Listener>');
   if iberr = EADR then writeln (' EADR <Address error>');
   if iberr = EARG then writeln (' EARG <Invalid argument>');
   if iberr = ESAC then writeln (' ESAC <Not Sys Ctrlr>');
   if iberr = EABO then writeln (' EABO <Op. aborted>');
   if iberr = ENEB then writeln (' ENEB <No GPIB board>');
   if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
   if iberr = ECAP then writeln (' ECAP <No capability>');
   if iberr = EFSO then writeln (' EFSO <File sys. error>');
   if iberr = EBUS then writeln (' EBUS <Command error>');
   if iberr = ESTB then writeln (' ESTB <Status byte lost>');
   if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
   if iberr = ETAB then writeln (' ETAB <Table Overflow>');

   writeln('ibcnt = ', ibcnt);
```

```
(*  Call the IBONL function to disable the hardware and       *)
(*  software.                                                 *)

   ibonl (dvm,0);

   halt;

end;

(* ========================================================
 *                      Procedure READDATA
 *  This procedures reads 10 measurements from the Fluke 45
 *  and calculates the average of the measurements.
 *
 *  The EXIT procedure terminates this procedure.
 * ========================================================*)
procedure readdata;

begin

(*  Initialize the accumulator of the 10 measurements to      *)
(*  zero.                                                     *)

   sum := 0.0;

(*  Establish FOR loop to read the 10 measuements.  The       *)
(*  variable m will serve as the counter of the FOR loop.     *)

   for m := 1 to 10 do
    begin

     (*  Trigger the Fluke 45.  If the error bit ERR is       *)
     (*  set in IBSTA, call GPIBERR with an error message.    *)

        ibtrg(dvm);
        if (ibsta and ERR) <> 0 then gpiberr('Ibtrg Error');

     (*  Request the triggered measurement by sending the     *)
     (*  instruction 'VAL1?'.  If the error bit ERR is set    *)
     (*  IBSTA, call GPIBERR with an error message.           *)

        wrt := 'VAL1?';
        ibwrt (dvm,wrt^,5);
        if (ibsta and ERR) <> 0 then gpiberr('Ibwrt Error');

     (*  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.           *)
```

```
        mask := $4800;                  (* RQS + TIMO *)
        ibwait(dvm, mask);
        if (ibsta and (ERR or TIMO)) <> 0 then
            gpiberr('Ibwait Error');

      (*  Read the Fluke 45 serial poll status byte.  If       *)
      (*  the error bit ERR is set in IBSTA, call GPIBERR      *)
      (*  with an error message.                               *)

        spr := 0;
        ibrsp(dvm,spr);
        if (ibsta and ERR) <> 0 then gpiberr('Ibrsp Error');

      (*  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 <> $50) then dvmerr('Fluke 45 Error', spr);

      (*  Read the Fluke 45 measurement.  If the error bit     *)
      (*  ERR is set in IBSTA, call GPIBERR with an error      *)
      (*  message.                                             *)

        ibrd (dvm, rd, 10);
        if (ibsta and ERR) <> 0 then gpiberr('Ibrd Error');

      (*  Remove spaces in array RD.  Print the measurement.   *)

        rd[ibcnt-1] := #0;
        writeln('Reading: ', rd);
        writeln;

      (*  Convert the measurement to its numeric value.        *)
      (*  If there is an error during the conversion, print    *)
      (*  the index of the character that did not convert      *)
      (*  and terminate this program.  If an error does not    *)
      (*  occur during the conversion, add the value to the    *)
      (*  accumulator.                                         *)

        val(rd, num, code);
        if code <> 0 then
         begin
           writeln('Error at position: ', code);
           exit;
         end
        else
         sum := sum + num;

    end;  (*  Continue FOR loop until 10 measurements are     *)
          (*  read.                                           *)

(*  Print the average of the 10 readings.                      *)
```

```
   writeln('The average of the 10 readings is ', sum/10);

end;

(* ========================================================
 *                          MAIN
 * ========================================================*)

BEGIN

    writeln('Read 10 measurements from the Fluke 45...');
    writeln;

(*  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.    *)

    dvm := ibfind ('DVM');
    if (dvm < 0) then gpiberr('Ibfind Error');

(*  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.                            *)

    ibclr (dvm);
    if (ibsta and ERR ) <> 0 then gpiberr('Ibclr Error');

(*  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.                                    *)

    wrt := '*RST; VAC; AUTO; TRIGGER 2; *SRE 16';
    ibwrt (dvm, wrt^, 35);
    if (ibsta and ERR) <> 0 then gpiberr('Ibwrt Error');

(*  Call READDATA to read 10 measurements from the            *)
(*  Fluke 45.                                                 *)

    readdata;

(*  Call the IBONL function to disable the device DVM.        *)

    ibonl (dvm,0);

end.
```

Turbo Pascal for Windows Program – Board Functions

```
PROGRAM BTPWSAMP(input,output);

uses wincrt, tpwgpib;

Var
  cmd  : PChar;                (* Commands written to         *)
                               (* the Fluke 45.               *)
  wrt  : PChar;                (* Data written to the         *)
                               (* Fluke 45.                   *)
  rd   : array[0..10] of char; (* Data received from the      *)
                               (* Fluke 45.                   *)
  mask : word;                 (* Wait mask.                  *)
  bd   : integer;              (* Board number.               *)
  i,m  : integer;              (* FOR loop index.             *)
  code : integer;              (* Procedure VAL parameter.    *)
                               (* VAL is a Turbo Pascal       *)
                               (* conversion procedure.       *)
  num  : real;                 (* Numeric conversion of RD.   *)
  sum  : real;                 (* Accumulator of              *)
                               (* measurements.               *)

(*=========================================================
 *                       Procedure DVMERR
 *  This procedure 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.
 *
 *  The HALT procedure will terminate this program.
 * =========================================================*)
procedure dvmerr(msg:string; rdchar:char);

begin

    writeln (msg);
    writeln('Status Byte = ', ord(rdchar));

(*  Call the IBONL function to disable the hardware and       *)
(*  software.                                                 *)

    ibonl (bd,0);

    halt;

end;
```

```
(* ========================================================
 *                       Procedure GPIBERR
 *  This procedure will notify you that a NI-488 function
 *  failed by printing an error message.  The status variable
 *  IBSTA will be printed in decimal 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 IBCNTL
 *  will be printed in decimal.
 *
 *  The NI-488 function IBONL is called to disable the
 *  hardware and software.
 *
 *  The HALT procedure stops execution of this program.
 * ========================================================*)
procedure gpiberr(msg:string);

begin

   writeln (msg);

   write('ibsta = ', ibsta,' <');
   if ibsta and ERR  <> 0 then write (' ERR');
   if ibsta and TIMO <> 0 then write (' TIMO');
   if ibsta and EEND <> 0 then write (' END');
   if ibsta and SRQI <> 0 then write (' SRQI');
   if ibsta and RQS  <> 0 then write (' RQS');
   if ibsta and CMPL <> 0 then write (' CMPL');
   if ibsta and LOK  <> 0 then write (' LOK');
   if ibsta and REM  <> 0 then write (' REM');
   if ibsta and CIC  <> 0 then write (' CIC');
   if ibsta and ATN  <> 0 then write (' ATN');
   if ibsta and TACS <> 0 then write (' TACS');
   if ibsta and LACS <> 0 then write (' LACS');
   if ibsta and DTAS <> 0 then write (' DTAS');
   if ibsta and DCAS <> 0 then write (' DCAS');
   writeln(' >');

   write('iberr = ', iberr);
   if iberr = EDVR then writeln (' EDVR <DOS Error>');
   if iberr = ECIC then writeln (' ECIC <Not CIC>');
   if iberr = ENOL then writeln (' ENOL <No Listener>');
   if iberr = EADR then writeln (' EADR <Address error>');
   if iberr = EARG then writeln (' EARG <Invalid argument>');
   if iberr = ESAC then writeln (' ESAC <Not Sys Ctrlr>');
   if iberr = EABO then writeln (' EABO <Op. aborted>');
   if iberr = ENEB then writeln (' ENEB <No GPIB board>');
   if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
   if iberr = ECAP then writeln (' ECAP <No capability>');
   if iberr = EFSO then writeln (' EFSO <File sys. error>');
   if iberr = EBUS then writeln (' EBUS <Command error>');
   if iberr = ESTB then writeln (' ESTB <Status byte lost>');
   if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
   if iberr = ETAB then writeln (' ETAB <Table Overflow>');
```

```
   writeln('ibcnt = ', ibcnt);

(*  Call the IBONL function to disable the hardware and       *)
(*  software.                                                 *)

   ibonl (bd,0);

   halt;

end;

(* ========================================================
 *                      Procedure READDATA
 *  This procedures reads 10 measurements from the Fluke 45
 *  and calculates the average of the measurements.
 *
 *  The EXIT procedure terminates this procedure.
 * ========================================================*)
procedure readdata;

begin

(*  Initialize the accumulator of the 10 measurements to      *)
(*  zero.                                                     *)

    sum := 0.0;

(*  Establish FOR loop to read the 10 measuements.  The       *)
(*  variable m will serve as the counter of the FOR loop.     *)

    for m := 1 to 10 do
     begin

      (*  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.                 *)

        cmd := '!@';
        ibcmd (bd, cmd^, 2);
        if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');

      (*  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.                                            *)

        cmd := #8;
        ibcmd (bd, cmd^,1);
        if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');
```

```
(*  Request the triggered measurement by sending the    *)
(*  instruction "VAL1?".  If the error bit ERR is       *)
(*  set IBSTA, call GPIBERR with an error message.      *)

  wrt := 'VAL1?';
  ibwrt (bd, wrt^,5);
  if (ibsta and ERR) <> 0 then gpiberr('Ibwrt Error');

(*  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.                      *)

  mask := $5000;                (* SRQI + TIMO *)
  ibwait(bd, mask);
  if (ibsta and (ERR or TIMO)) <> 0 then
      gpiberr('Ibwait Error');

(*  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 r 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.                                            *)

  cmd := '_?'#24'A ';
  ibcmd(bd, cmd^, 5);
  if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');

(*  Read the Fluke 45 serial poll status byte.  If      *)
(*  the error bit ERR is set in IBSTA, call GPIBERR     *)
(*  with an error message.                              *)

  ibrd(bd, rd, 1);
  if (ibsta and ERR) <> 0 then gpiberr('Ibrd 1 Error');

(*  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[0]) <> $50) then dvmerr('Fluke 45 Error',
      rd[0]);
```

```
      (*  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.                                            *)

        cmd := #25'A ';
        ibcmd(bd, cmd^, 3);
        if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');

      (*  Read the Fluke 45 measurement.  If the error bit *)
      (*  ERR is set in IBSTA, call GPIBERR with an error     *)
      (*  message.                                            *)

        ibrd (bd,rd,10);
        if (ibsta and ERR) <> 0 then gpiberr('Ibrd 2 Error');

      (*  Assign the array RD to the string BUFFER.           *)
      (*  Remove spaces in BUFFER.  Print the measurement.    *)

        rd[ibcnt-1] := #0;
        writeln('Reading: ', rd);
        writeln;

      (*  Convert the measurement to its numeric value.       *)
      (*  If there is an error during the conversion,         *)
      (*  print the index of the character that did not       *)
      (*  convert and terminate this program.  If an error    *)
      (*  does not occur during the conversion, add the       *)
      (*  value to the accumulator.                           *)

        val(rd, num, code);
        if code <> 0 then
          begin
          writeln('Error at position: ', code);
          exit;
          end
        else
          sum := sum + num;

     end;  (*  Continue FOR loop until 10 measurements        *)
           (*  are read.                                       *)

(*  Print the average of the 10 readings.                      *)

    writeln('The average of the 10 readings is ', sum/10);

end;
```

```
(* ======================================================
 *                    MAIN
 * ======================================================*)

BEGIN

    writeln('Read 10 measurements from the Fluke 45...');
    writeln;

(*  Assign a unique identifier to board 0 and store in        *)
(*  the variable BD.  The name 'GPIB0' is the default name    *)
(*  of board 0.  If BD is less than zero, call GPIBERR        *)
(*  with an error message.                                    *)

    bd := ibfind ('GPIB0');
    if (bd < 0) then gpiberr('Ibfind Error');

(*  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.                                                  *)

    ibsic (bd);
    if (ibsta and ERR) <> 0 then gpiberr('Ibsic Error');

(*  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.                *)

    ibsre (bd,1);
    if (ibsta and ERR) <> 0 then gpiberr('Ibsre Error');

(*  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.                            *)

    cmd := #17#20'!@';
    ibcmd (bd, cmd^,4);
    if (ibsta and ERR) <> 0 then gpiberr('Ibcmd Error');

(*  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.                       *)

    wrt := '*RST; VAC; AUTO; TRIGGER 2; *SRE 16';
    ibwrt (bd,wrt^,35);
    if (ibsta and ERR) <> 0 then gpiberr('Ibwrt Error');

(*  Call READDATA to read 10 measurements from the            *)
(*  Fluke 45.                                                 *)

    readdata;

(*  Call the IBONL function to disable the hardware and       *)
(*  software.                                                 *)

    ibonl (bd,0);

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 explains how to use, install, and configure the Applications Monitor, a resident program that is useful in debugging sequences of NI-488 and NI-488.2 calls from within your MS-DOS application.

Using the Applications Monitor

The applications monitor can temporarily halt program execution (trap) upon returning from NI-488 functions and NI-488.2 routines that meet a condition specified by you. You 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 or NI-488.2 routine, 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 specified 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 revisions of the NI-488.2 software for MS-DOS.

Installing the Applications Monitor

The applications monitor is included on the distribution diskette as the file `APPMON.EXE`. To install it, enter the following command in response to the DOS prompt:

```
APPMON
```

If the GPIB driver is not present or the applications monitor has already been installed, the file does not load and an error message is printed.

Once installed, the applications monitor will remain in memory until you restart the system. If you decide later 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 the DOS prompt using the special utility program called `IBTRAP.EXE`.

With the function call and the DOS utility, 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 Chapter 3 of this manual for details about how to include `ibtrap` calls in your application.

You can use the utility program `ibtrap` to set the trap mode from DOS. Simply enter `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 flag will leave its current configuration unchanged. Invoking `ibtrap` without any flags will display the valid flags and their current states. 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 relative to the current GPIB call:

- `Device` symbolic device name.
- `Function` NI-488.2 routine or NI-488 function 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>. 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. A pop-up menu appears with each of the status bits displayed along with their current states (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. A pop-up menu appears 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>within the applications monitor, with the program active, you can view program output written to the screen. 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 technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *Technical Support Form* before contacting National Instruments helps us help you better and faster.

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

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

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

Technical Support Form

Technical support is available any time by fax. 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 _____MB

Display adapter _____

Mouse _____yes _____no

Other adapters installed _____

Hard disk capacity _____MB Brand _____

Instruments used _____

National Instruments hardware product model _____

Revision _____

Configuration _____

(continues)

National Instruments software product _____

Version _____

Configuration _____

The problem is _____

_____

_____

_____

_____

_____

_____

_____

List any error messages _____

_____

_____

_____

_____

_____

_____

_____

The following steps will reproduce the problem _____

_____

_____

_____

_____

_____

_____

Pascal 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 _____
- Application Programming Language/Version (IBM Pascal, Microsoft Pascal, MS QuickPascal, Turbo Pascal, Turbo Pascal for Windows) _____
- Programming Language Interface Version _____
- 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 _____
- 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 Pascal**

Edition Date: **November 1993**

Part Number: **320375-01**

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