

ARTISAN®
TECHNOLOGY GROUP

Your **definitive** source for quality pre-owned equipment.

Artisan Technology Group
(217) 352-9330 | sales@artisantg.com | artisantg.com

Full-service, independent repair center
with experienced engineers and technicians on staff.

We buy your excess, underutilized, and idle equipment
along with credit for buybacks and trade-ins.

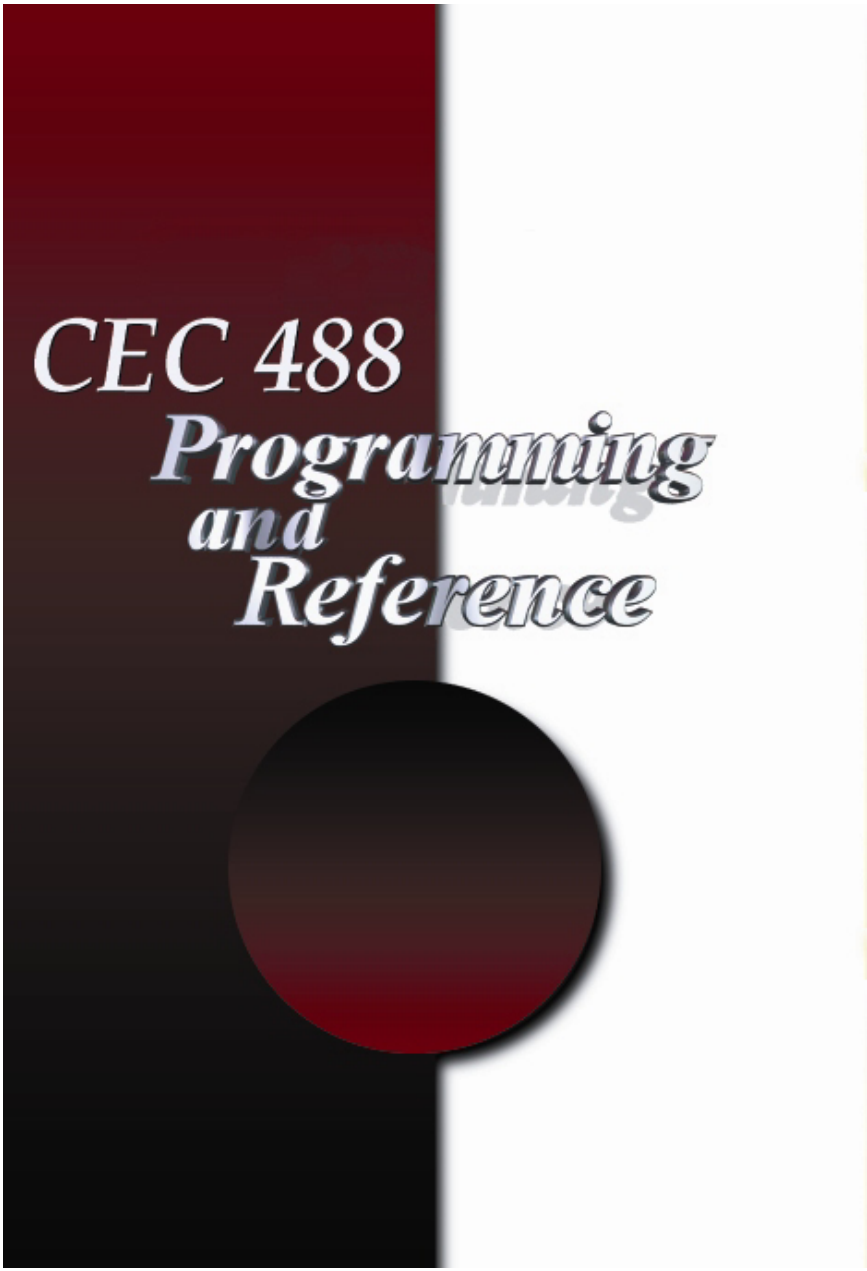
Custom engineering
so your equipment works exactly as you specify.

- Critical and expedited services
- In stock / Ready-to-ship
- Leasing / Rentals / Demos
- ITAR-certified secure asset solutions

Expert team | Trust guarantee | 100% satisfaction

All trademarks, brand names, and brands appearing herein are the property of their respective owners.

Find the *Measurement Computing / CEC PC-488* at our website: *Click HERE*


CEC 488
Programming
and
Reference

Program and documentation copyrighted 1986, 1998, 2003 by Capital Equipment Corporation (CEC). The software interpreter contained in EPROM/ROM is copyrighted and all rights are reserved by Capital Equipment Corporation. Copying or duplicating this product is a violation of law.

Application software libraries provided on disk are copyrighted by Capital Equipment Corporation. The purchaser is granted the right to include portions of this software in products which use one of CEC's IEEE-488 interface boards (including those sold through resellers such as Keithley Instruments, etc.). The software may not be distributed other than for the application just mentioned.

Purchasers of this product may copy and use the programming examples contained in this book. No other parts of this book may be reproduced or transmitted in any form or by any means, electronic, optical, or mechanical, including photocopying and recording, or by any information storage and retrieval system, without permission in writing from Capital Equipment Corporation.

For CEC contact information, please go to `http://www.cec488.com` on the World Wide Web

Part number 370966A-01
November 2003

All products are warranted against defective materials and workmanship for a period of one year from the date of delivery to the original purchaser. Any product that is found to be defective within the warranty period will, at the option of the manufacturer, be repaired or replaced. This warranty does not apply to products damaged by improper use.

CEC and CEC resellers assume no liability for damages consequent to the use of this product. This product is not designed for use in life support applications.

Information furnished by the manfacturer is believed to be accurate and reliable. However, CEC and CEC resellers assume no responsibility for the use of such information nor for any infringements of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent rights of Keithley Instruments or CEC.

Table of Contents

Introduction and Installation 1-1

- Step 1: Install the Software ... 1-3
- Step 2: Install the Hardware ... 1-5
- Step 3: Test the Interface ... 1-6
- Step 4: Applications ... 1-8

IEEE-488 Tutorial 2-1

- GPIB Concepts ... 2-2
- About the IEEE-488 Standard ... 2-4
- GPIB Device Addresses ... 2-5
- Controllers, Listeners and Talkers ... 2-6
- Data Transfers ... 2-7
- GPIB Commands ... 2-8

Programming IEEE-488 Interfaces 3-1

- Beginning your program ... 3-2
- Initializing the GPIB ... 3-5
- Sending Data to a Device ... 3-6
- GPIB Addresses - Primary and Secondary ... 3-7
- Reading Data from a Device ... 3-8
- Checking the Device Status - Service Request ... 3-13
- Testing for the Presence of a Device ... 3-16
- Testing for the Presence of the GPIB Board ... 3-17
- Checking Hardware Features of the GPIB Board ... 3-18
- Low-level GPIB Control using Transmit ... 3-20
- Sending a Device Clear Command ... 3-32
- Sending a Device Trigger Command ... 3-33
- The Receive Routine ... 3-34
- Binary Data Transfers ... 3-36
- Data Formats ... 3-39
- High Speed Data Transfers ... 3-40
- Configuring Board Parameters ... 3-42
- Configuration Files ... 3-47

Advanced Programming 4-1

- The Computer as a GPIB Device ... 4-2
- Passing Control ... 4-18
- Interrupt Processing ... 4-19
- Using Multiple Boards ... 4-24

Programming Examples 5-1
BASICA/GWBASIC Examples ... 5-3
Turbo Pascal Examples ... 5-8
C Examples ... 5-12

Technical Reference 6-1
Introduction ... 6-2
Electrical Specifications ... 6-4
Handshake Timing Sequence ... 6-5
Mechanical Specifications ... 6-7
Bus Line Definitions ... 6-8
Cabling Information ... 6-9

BASICA and GWBASIC Language Interface A-1
IEEE-488 Subroutine calls ... A-2
Examples ... A-8

QuickBASIC, QBASIC, and Visual BASIC Language Interface B-1
IEEE-488 Subroutine calls ... B-3
Using Tarray and Rarray with strings in DOS QuickBASIC ... B-9
Examples - QuickBASIC or QBASIC ... B-10
Examples - Visual BASIC ... B-11

Turbo Pascal and Delphi Language Interface C-1
IEEE-488 Subroutine calls ... C-2
Examples ... C-7

C and C++ Language Interface D-1
IEEE-488 Subroutine calls ... D-3
Examples ... D-8

FORTRAN Language Interface E-1
IEEE-488 Subroutine calls ... E-3
Examples ... E-9

Assembler and other language interfacing F-1
IEEE-488 Subroutine calls ... F-2
Examples ... F-10

OS/2 and IEEE-488 G-1

Troubleshooting I-1

Checklist for Solving 488 Programming Problems I-2

Hardware Configuration J-1

PC488 ..... J-2
PC488 (revision C and earlier) J-6
488EX (16-bit ISA bus card) J-11
PCI488 (PCI bus board) J-14
PS488 (Microchannel, PS/2 board) J-15

Differences from earlier versions K-1

ASCII character table & GPIB codes L-1

Using PRINT and INPUT for GPIB control M-1

HP-style universal language driver O-1

Using the CECHP driver O-2
Converting old HP computer programs to use CECHP O-3
Commands O-5
Examples O-10
Accessing your board from Spreadsheets O-13
Reading binary data O-14

IEEE-488.2 P-1

488SD: High Speed Streaming Data Protocol Q-1

How 488SD Works Q-2
Using 488SD on the 488EX board Q-4
488SD Technical Data Q-7

I wish he would explain his explanation.
- Lord Byron (1788-1824)

He can compress the most words into the smallest idea of any man I've ever met.
- Abraham Lincoln (1809-1865)

Your interface board consists of hardware and software that fully implement the IEEE-488 standard, also known as GPIB. This interface is an international standard that allows the PC to communicate with over 2000 instruments made by over 200 manufacturers.

There are multiple models of IEEE-488 interface: 8- and 16-bit ISA bus cards, a PCI bus card, a PS/2 Microchannel version. Although installation differs for these products, programming is identical.

Your board can be used with your own programs in any of the popular programming languages for instrument control applications. It can also be used without any programming to connect HPIB peripheral devices such as printers and plotters to the PC.

Your board's key features include:

- Implementation of the entire IEEE-488 standard. Your board can operate as a system controller or a device.

- A choice of programming methods: subroutine CALLs for maximum speed and flexibility, or a device driver that may be accessed very simply using file input/output statements.

- High-speed data transfers.

- Support of HPIB printers and plotters for use with MS-DOS or Windows wordprocessing, spreadsheet, and graphics programs.

Note: The IEEE-488 reference document may be ordered by writing to the IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854

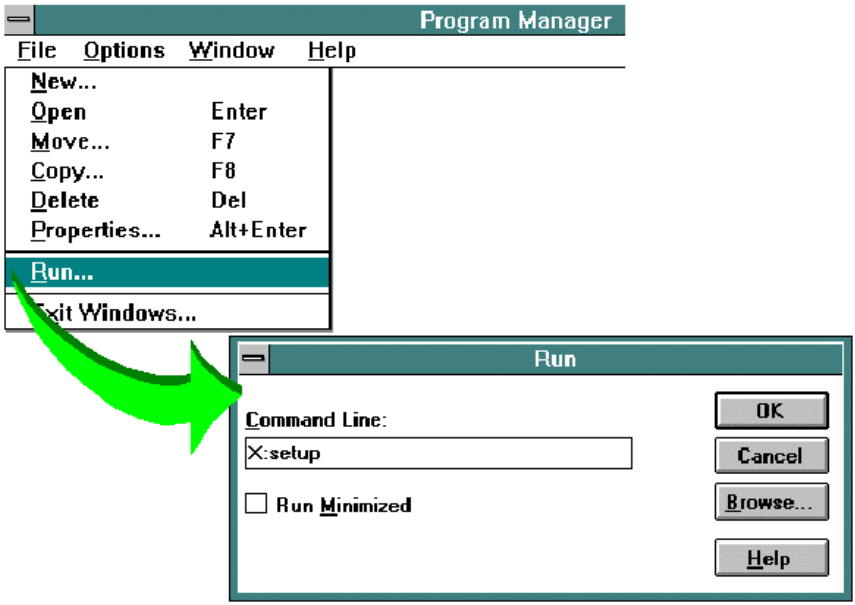
Step 1: Install the Software

Run the **SETUP** program from the disk supplied with the interface card.

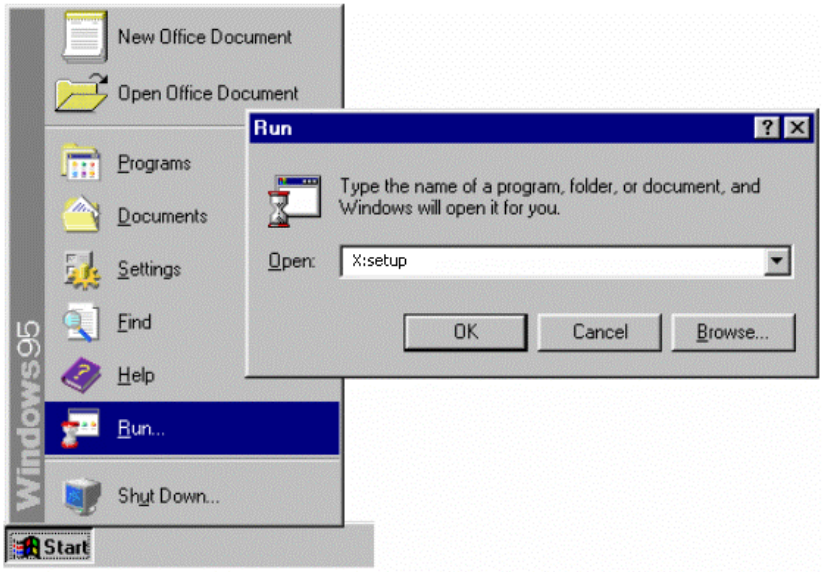
For DOS, just type the program name, where *X* equals your CD drive:

```
X:SETUP
```

For **Windows 3.x**, use the Program Manager's File/Run menu, where *X* equals your CD drive, as shown in the following illustration:



For later versions of Windows (**Win95, WinNT**...), use the Start button and choose Run. In the following illustration *X* equals your CD drive:



Note: when installing in Windows NT, you must be logged in as the Administrator, to have appropriate privileges for installing device drivers.

You'll be asked to choose a directory for software installation, and to select which components you want to install. The default is to install everything, but you can choose to leave out support for programming languages you won't be using.

Remove the interface board from its protective packaging by grasping the metal rear panel. Save the anti-static bubble package.

The board should be handled only by the edges. The integrated circuits on the board can be damaged by static electric discharge.

Configuration

- Some board models (8- and 16-bit ISA bus boards) have switches and jumpers to select hardware configuration. **In most cases, there will be no need to change any switch settings.** If you do find a conflict with other boards in your computer, see the Hardware Configuration section for information on settings for your particular board model.
- Other board models, such as PCI boards, have no switches. **These are plug-and-play boards, which configure automatically.**

Turn the PC power OFF. Unplug the power cord.
Remove the cover of the PC.

Install the board in any available slot.

Note: 16-bit ISA bus cards must be installed in a 16-bit slot.
Note: PCI bus cards must be installed in a PCI slot.

Use the screw to attach the rear panel bracket to the computer case.
Reinstall the computer's cover and plug it back in.

Turn the computer on.

Note: for PCI cards in a plug-and-play operating system like Win95, the card should configure automatically, if you have installed the software FIRST. If you get any message about an unrecognized board and a prompt for a disk, insert the software diskette as instructed by the operating system.

First, run the hardware test program TEST488, from a command-line prompt:

```
C:\CEC488>  test488_
```

This will test and report on the status of your IEEE-488 board(s).
If you have any errors, or the board is not recognized, see the section on Troubleshooting.

Next, connect an IEEE-488 device to the interface, and run the interactive program TRTEST, which allows you to try to communicate with the device:

```
C:\CEC488>  trtest_
```

Choose **Send** from the menu (by typing **S**), and enter the address of your device, then a valid command string for the device.
Any IEEE-488.2 compatible device will accept the command ***IDN?**
(note that not all devices support the 488.2 extensions)

```
CEC488 Interactive Utility, v2.0.  Copyright(C) 1997 CEC.

  Main functions   Configuration     Inquiry

  I)  Initialize   P)  SetPort       BP)  BoardPresent?
  S)  Send         ST) SetTimeout    LP)  ListenerPresent?
  E)  Enter        SI) SetInputEOS   F)   Feature?
  SP) Spoll        SO) SetOutputEOS  SRQ) SRQ?
  PP) Ppoll        B)  BoardSelect
  T)  Transmit     D)  DmaChannel
  R)  Receive
  TA) Tarray
  RA) Rarray

  Q)  Quit

  Choose a function: S
  GPIB address: 16
  Data: *IDN?

  Status:  0


  Hit Enter to continue...
```

You should see a status of 0, indicating everything is OK.

Then, if a reply from the instrument is expected (note that this depends on the device and the command string you have sent), you can use the **Enter** command to receive the response:

```
CEC488 Interactive Utility, v2.0.  Copyright(C) 1997 CEC.

  Main functions    Configuration      Inquiry

  I)  Initialize    P)  SetPort        BP)  BoardPresent?
  S)  Send          ST) SetTimeout     LP)  ListenerPresent?
  E)  Enter         SI) SetInputEOS    F)   Feature?
  SP) Spoll         SO) SetOutputEOS   SRQ) SRQ?
  PP) Ppoll         B)  BoardSelect
  T)  Transmit      D)  DmaChannel
  R)  Receive
  TA) Tarray
  RA) Rarray

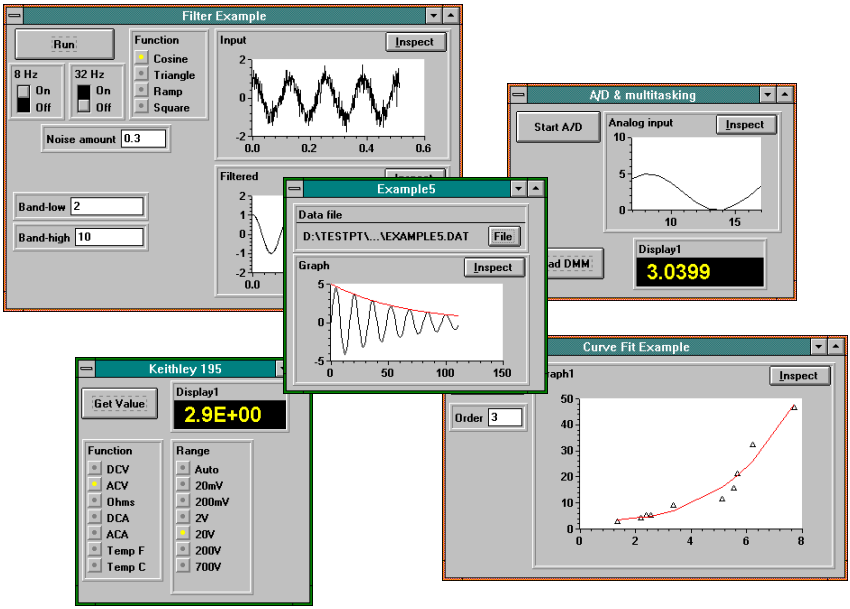
  Q)  Quit

  Choose a function: E
  GPIB address: 16
  Maxlen: 80

  Len: 56
  Data: KEITHLEY INSTRUMENTS INC.,MODEL 2001, 518567,A01  /A01
  Status:  0

  Hit Enter to continue...
```

Using your interface with TestPoint™



TestPoint is a software tool for building test, measurement and data acquisition applications for Windows. TestPoint lets you build complete applications without drawing, connecting, or wiring icons or writing lines of code. TestPoint brings extraordinary capability to instrument control and data acquisition for the benchtop or production line. Through the use of TestPoint's sharp graphics and clear indicators you can eliminate test and measurement errors and improve accuracy. TestPoint includes features that allow you to "hot link" data to your spreadsheets, databases, and word processing files so that the paperwork is done the instant the test is finished.

Your IEEE-488 interface board is directly supported by TestPoint's built-in GPIB object and wide selection of instrument libraries.

For additional information on TestPoint call 1-800-234-4232 or 1-800-348-0033.

Using your interface for writing custom programs

To use your interface for custom applications, you will want to write specialized software. We've made this easy by providing a simple set of high-level routines that may be used with all of the popular programming languages.

If you have no IEEE-488 experience, you may wish to read the tutorial in section 2.

Your IEEE-488 interface may be programmed in two different ways: using subroutine calls, or a device driver. The first method takes full advantage of the speed and power of the interface. The second method is very simple to use, but limited in speed. (Note also: the device driver method requires that you use DOS, Win 3.x, or Win95 - an operating system that can load DOS-style device drivers).

Generally, you should use the subroutine calling method of programming, as long as your programming language is supported (C, most versions of BASIC, Pascal or Delphi, many FORTRAN versions, etc.).
Use the device driver method (also called "Universal Language interface") for other languages and programming environments, such as spreadsheet macro languages.

NOTE: if you are trying to run an older, existing program on a new card, check the hardware configuration section of this manual and the section on differences between versions of software (section K).
In particular, if you need to force the I/O address of a PCI-bus card, you can use the PCICONF utility (see section J or K).

If you intend to use your interface with GPIB peripherals such as HP printers and plotters, from DOS or Windows, no programming is necessary. All you need is the interface card and the PRINTER and/or SERIAL utility programs provided on the applications disk.

PRINTER allows you to make GPIB devices look like standard parallel printer ports (LPT1, 2 or 3). You can install GPIB devices or parallel printers in any combination. To install a GPIB printer at address one so that it appears as LPT1, use the command:

```
C:\CEC488>  utility\printer  1
```

Up to three values can follow PRINTER on the command line. These values specify the devices that will appear as LPT1, 2 and 3. Each value can be either a GPIB address (from 0 to 30), or P1, P2, or P3, to specify one of the actual parallel ports on your PC.

SERIAL allows you to make GPIB devices look like standard serial ports. You can install GPIB devices and serial devices in any combination. To install a GPIB plotter at address five so that it appears as COM1, use the command:

```
C:\CEC488>  utility\serial  5
```

Up to two values can follow SERIAL on the command line. These values specify the devices that will appear as COM1 and COM2. Each value can be either a GPIB address (from 0 to 30), or S1 or S2 to specify one of the actual serial ports.

You can put the PRINTER or SERIAL command lines in your AUTOEXEC.BAT file if you wish to have them execute automatically whenever you reboot your computer.

Using your interface with other (third-party) software packages

Your IEEE-488 interface board is also compatible with many existing software packages for instrument control, including instrument-specific programs from many instrument manufacturers, and other tools such as LabView or LabWindows. The compatibility drivers required for some of these programs are installed as part of the normal SETUP program.

A little learning is a dangerous thing;
Drink deep, or taste not the Pierian spring.
- Alexander Pope (1688-1744)

You can observe quite a lot just by watching.
- Yogi Berra

The main purpose of the general purpose-interface bus (GPIB) is to send information between two or more devices. Before any data is sent, the devices must be configured to send the data in the proper order and according to the proper protocol.

An appropriate analogy for the organization of the GPIB is a New England town meeting. New England town meetings are similiar to most committee meetings; however, they require a stern moderator to maintain control over the representatives and to enforce the rules of order during the meeting. If the moderator cannot attend a meeting, he may appoint a replacement who has most but not all of the power of the elected moderator.

The moderator of a GPIB system is the system controller. The system controller determines which device talks and when it can talk. The system controller can also appoint a replacement, which then becomes the active controller.

In any hotly debated issue it is the moderator's responsibility to insure that only one person speaks at a time. This is also the active controller's responsibility in that it must recognize which single device may talk on the bus. Of course not all issues are hotly debated and some of the representatives may not care to listen or fall asleep. In a GPIB system the active controller can define which devices will listen if the information is not required by all of the devices on the bus.

If a representative falls asleep it is the moderator's responsibility to wake him up. The active controller wakes up inactive listeners by asserting attention and sending bus commands that specify the new talker and listeners. It is the assertion of attention that establishes the fact that the data on the bus represents an interface command.

When attention is not asserted, the data on the bus represents a message from a talker to a listener.

In a complex debate there may be several obscure rules of order invoked. These rules are infrequently used but necessary in special situations. The same is true for a GPIB system. While most of the data transfer is routine between talkers and listeners, occasionally a special command is required to perform a specific function.

If the moderator has done his job properly, the town meeting will end with everyone shaking hands. The GPIB is a little friendlier in this respect in that it "shakes hands" during every data transaction. The purpose of the hardware handshake is to insure that no device on the bus misses any information. This protocol allows the GPIB to accommodate both fast and slow listeners because they can receive data from the same source. The disadvantage is that the data rate is controlled by the slowest listener.

When programming any device on the bus it is helpful to remember the town meeting analogy.

Moderator	-- System controller or active controller
Meeting members	-- Devices on the bus
Talker	-- Talker or data source
Listener	-- Listener or data receiver
Rules of order	-- Commands and functions
Social graces	-- Hardware handshake

- Any number of devices can listen but all may not be interested.
- Only one device can talk at a time or the messages would be confusing.
- There can only be one moderator at a time but he can designate another to take his place.
- A talker usually doesn't listen to himself.

The IEEE-488 standard defines the electrical specifications as well as the cables, connectors, control protocol, and messages required to allow information transfer between devices. The Institute of Electrical and Electronic Engineers (IEEE) adopted in 1975 a proposal made by Hewlett-Packard for instrumentation control. Revisions were published in 1978 and 1980.

Up to 14 devices may be connected to a computer by chaining IEEE-488 cables from one device to the next. A single IEEE-488 interface can control all these devices. The standard specifies that up to 20 meters of cable can be used, or 2 meters times the number of devices, whichever is less. Data may be transferred at up to 1M bytes/second, if the devices are designed for that speed.

IEEE-488 ensures that devices can communicate, but, in general, does not define the character sequences that make a device carry out a particular operation. That is up to the manufacturer. For example, a Keithley voltmeter might use "R3X" to set its voltage range, while another company's meter might use "RANGE=100V".

In 1987, additional standards were adopted for the format of device data and commands. The original standard, covering hardware and electrical details, was renamed IEEE-488.1, and the new standard covering message formats was named IEEE-488.2. The 488.2 standard describes the structure of commands and data responses, as well as how to handle some error conditions. However, the command strings are still left up to the device manufacturer.

In 1990, the first proposal for SCPI (Standard Commands for Programmable Instrumentation) was published. SCPI is a set of common commands intended to allow instruments from different manufacturers to be controlled in a consistent fashion. The SCPI committee is still working out many of the details.

In 1992, Capital Equipment Corporation proposed the 488SD specification to break the 1MB/sec barrier and allow very high speed data transfers. 488SD allows a maximum rate of 5M bytes/second, depending on cable lengths. 488SD devices are compatible with existing 488 systems.

Each device in a GPIB system has an address, which is a number between 0 and 30.

This includes the controller, which is often assigned address 21 (although any address is valid). All device addresses in a GPIB system must be unique.

When communication occurs between devices, addresses are used to make sure that the desired devices receive the information and that other devices ignore it.

Device addresses are often set by a switch on the device. Once the switch is set, you should usually cycle power to the device to ensure that it is using the new address setting. Some devices do not have a separate switch, but instead allow their addresses to be set from front panel keys. The GPIB address of your interface boards is set in software.

The GPIB addresses just described are also known as **primary addresses**.

This is because some devices also have what are known as **secondary addresses**, which select a particular functional block within the device. For example, an oscilliscope with plug-in signal conditioning modules may have a primary address of 3. The first plug-in module may have a secondary address of 1, and is therefore referenced on the GPIB system by using the combination of its primary and secondary addresses.

No man would listen to you talk if he didn't know it was his turn next. -Ed Howe

Don't talk unless you can improve the silence. -Laurence C. Coughlin

In a GPIB system, there is a single **system controller**. This is usually the computer. The system controller oversees all operations on the GPIB.

The system controller can hand over control responsibilities to another device (if that device has control capability). This new device becomes the **active controller**. The system controller can always regain control by sending out an interface clear command.

The controller may assign other devices to transfer data by designating a **talker** and one or more **listeners**. Talkers and listeners are assigned by sending talk and listen commands on the GPIB. These commands include the addresses of the devices. There are also unlisten (UNL) and untalk (UNT) commands, to disable listeners and talkers.

Once a talker and some listeners have been assigned, a data transfer can start. The controller allows the transfer to begin by releasing the ATN (attention) signal on the GPIB. This signal is asserted by the controller when it is sending out GPIB commands like talk and listen addresses, and released when device-dependent data transfers are to occur.

The IEEE-488 standard uses "data transfer" to include not only measurement results, but any character sequences sent between devices. Instrument mode settings, such as "MODE ACV", are treated as data by the GPIB.

Data transfers are usually terminated by agreed-upon character sequences, such as carriage return and line feed. The special GPIB signal EOI is another way to mark the end of a data transfer. This signal may be asserted by the talker along with the last byte in a transfer.

In addition to device dependent character data transfers, GPIB defines a number of specialized commands that may be used by the controller. These commands are sent out with the ATN signal asserted as true, to differentiate them from data transfers.

The most commonly used commands are listed below. Those marked as **addressed** affect only those devices currently assigned as listeners. Commands marked as **universal** affect all devices.

Interface Clear	IFC	universal	resets GPIB activity
Device Clear	DCL	universal	resets device modes
Selected Dev. Clr.	SDC	addressed	resets device modes
Trigger	GET	addressed	starts device operation

Programming IEEE-488 Interfaces

Get your facts first, and then you can distort them as much as you please.
- Mark Twain

This section will show you how you can program your board for instrument control applications.

Examples in this section and the following section are given in the programming languages most popular with our customers: BASIC (Visual- or Quick-), Pascal or Delphi, and C.
Your board can be programmed in other languages using the same steps presented in this section. The reference sections of this manual give detailed information on the use of each language: the support files you will need, rules for writing code, instructions on compiling and running programs, and examples.

New languages may be available on the applications disk that are not listed in the manual. Insert the applications disk and type the file INDEX.DOC to see a complete list of all supported languages.

Over the next few pages, you'll see how to build a complete instrument control program using the **Initialize**, **Send**, and **Enter** commands. Other commands, for more sophisticated GPIB control, are introduced later.

Depending on your programming language, you need to begin each program with a few steps to tell the computer about the IEEE-488 interface subroutines.

QuickBASIC 4.x

You will need the files **IEEEQB.QLB, IEEEQB.LIB** and **IEEEQB.BI**. For convenience, you should copy these files to the working directory you will be using to develop your program.

When starting QuickBASIC, you must tell it to load the IEEE-488 routines by using a command line:

```
QB  /L  ieeeqb.qlb
```

Then, as the first line in your program, add the following line of code:

```
'$INCLUDE: 'ieeeqb.bi'
```

(Note: for earlier versions of QuickBASIC, see the QuickBASIC appendix - some functions may not be available.)

QBASIC

First, you must load the driver BASIC488, by running this command (the program is located in the BASIC subdirectory):

```
BASIC488
```

(Note: you may want to put this command in your AUTOEXEC.BAT file so it runs every time you boot your computer, so the driver is loaded automatically).

Then, run QBASIC and load the file IEEEQBAS.BAS (in the QBASIC subdirectory). This file includes the necessary program lines to allow you to use the IEEE-488 subroutines.

Choose File/Save As and save under a new program name.

Visual BASIC (including Visual BASIC for Application)

In Visual BASIC, use the File/Add File menu command and add the file **IEEEVB.BAS** to your project. This file is located in the VB subdirectory. This file contains all the subroutine declarations, so you can use the IEEE-488 routines in your program.

For VBA (Visual BASIC for Applications), which is built into products such as spreadsheets and word processors, you can simply insert the text of the IEEEVB.BAS file (or the older IEEEVB3.BAS file), and then call the routines.

BASICA, GWBASIC (obsolete versions of interpreted BASIC)

See Appendix A of this manual for instructions and examples.

C or C++ (any vendor)

You will need the file **IEEE-C.H**. (from the C subdirectory) You may wish to copy this file to the working directory you will be using to develop your program.
Put the following code line at the top of your program:

```
#include "ieee-c.h"
```

When you compile and link your program, you will need to provide the appropriate library, depending on your operating system:

for DOS, use:	IEEE488.LIB
for 16-bit Windows programs, use:	WIN488.LIB
for 32-bit Windows programs, use:	IEEE_32M.LIB (for Microsoft), or IEEE_32B.LIB (for Borland)
for 16-bit OS/2 programs, use:	IEEEOS2.LIB

Most modern C compilers have a development environment where you add files to a "project" or "workspace" screen to include them in your program. The details of adding files to a project varies from one compiler to another, and between versions - consult your compiler documentation.

Turbo Pascal

You will need the files **PAS488.OBJ** and **IEEEPAS.PAS**, both of which are provided in the TURBOPAS subdirectory.

Load IEEEPAS.PAS into Pascal and compile it (with destination set to "Disk") to build a Turbo Pascal "unit" file (.TPU).

Then, in writing your programs, make use of this interface unit file by adding the following line of code:

```
uses ieeepas;
```

Borland Delphi

You will need the file **IEEEDEL.PAS**, which is provided in the Delphi subdirectory. Add this file to your Delphi project.

In any module where you wish to use the IEEE-488 interface routines, add the following line of code:

```
uses ieeedel;
```

Other languages

See the specific language-interface appendices in the latter section of this manual, or check for .DOC files on the applications disk.

If your language is not directly supported for calling the interface routines, and you are using DOS or Win 3.x, you can use the Universal Language driver, see Appendix O.

Initializing the GPIB

The first step in any GPIB control program is to initialize the system. This is done with the **Initialize** routine.

The **Initialize** routine is called with two arguments: the first gives the GPIB address you wish to assign to the board, and the second specifies whether the board should be a system controller. The calling information is summarized below:

INITIALIZE (my.address,level)

where:

- my.address is an integer from 0 to 30, giving the GPIB address to be used by your board. This address should be different from the address of all devices connected to the computer.
- level is 0 to specify system controller, and 2 to specify device mode.

Initialize does a number of things when it is called. First, it prepares the interface for operation. Second, it sets the GPIB address to be used by your board. Third, if system control is specified, it sends out an interface clear (IFC) to the entire GPIB system, to initialize the other devices.

Note: If you are using PC488 **rev. C or earlier**, switch S1 position 8 is set at the factory to the off position to make PC488 a system controller. This allows PC488 to send GPIB bus commands. If you want PC488 to be a device (receive commands only) S1 position 8 should be ON. On all other models of 488 hardware, system controller function is handled in software.

BASIC (QuickBASIC, QBASIC, or Visual BASIC):

```
CALL INITIALIZE (21,0)
```

Turbo Pascal or Delphi:

```
initialize (21,0);
```

C or C++:

```
initialize (21,0);
```

Once the GPIB system has been initialized, the next step is usually to send a command or data to a device. The **Send** routine makes this easy:

SEND (address,info,status)

where:

- address is an integer set to the GPIB address of the destination device (0 to 30 for primary addresses - see next page for secondary addressing).
- info is a string to be sent to the device. Note: terminating character(s) are automatically added to the end of this string when it is sent. The default terminator is a line feed character. The terminator can be changed with the SetOutputEOS subroutine (see later in this section).
- status indicates whether the transfer went OK.
  0=OK
  8=timeout (instrument not responding)

This is the first time the **status** argument has appeared. It is generally the last argument in all board calls. Status will return with the value zero if the transfer went okay. The most common non-zero value for status is eight, which indicates a timeout. A timeout occurs if the device did not transfer data or the device took longer than the timeout period to send data. **Send** has only two status values.

BASIC (QuickBASIC, QBASIC, or Visual BASIC):

```
CALL SEND (7,"READ?",status%)
```

Turbo Pascal or Delphi:

```
send (7,'READ?',status);
```

C or C++:

```
send (7,"READ?",&status);
```

Most devices have only a **primary** GPIB address, which is a number from 0 to 30, which you will pass to routines like **Send** or **Enter**.
For example:

```
send (16,"*IDN?",&status);
enter (r,80,&l,16,&status);
```

However, some devices contain submodules with **secondary** addresses. You can access these devices by combining the primary and secondary addresses by the following formula:

100 * primary + secondary

for example, to access primary address 5, secondary address 20, you would use a value of 520:

```
send (520, "read?", &status);
```

These address values can be used with any of the routines that take device address parameters: **Send**, **Enter**, or **Spoll**.

Once an instrument has taken a measurement, the next step is to read the data into the computer. The **Enter** routine is used whenever you want to read from a device.

ENTER (recv,maxlength,length,address,status)

where:

- recv is a string variable which will contain the received data. **Enter** will terminate reception of data when: 1) the string is full, 2) a line feed is received, or 3) any character is received with the EOI signal. Carriage returns in the incoming data are ignored, and not placed in recv.
- maxlength is a value specifying the maximum number of characters you wish to receive. In DOS QuickBASIC, this argument is not present. maxlength can be a number from 0 to 65535 (hex FFFF).
- length will contain the actual number of characters received.
- address is the GPIB address of the device to read from.
- status indicates whether the transfer went OK (0=okay, 8=timeout)

After **Enter** finishes execution, the string variable will contain the received data. The number of bytes received will be returned in the length argument.

QuickBASIC or QBASIC (note: the maximum length is specified by pre-allocating a string using the SPACE$ function of BASIC):

```
r$=SPACE$(30)          ' allocate room for data
CALL ENTER (r$,length%,7,status%)
r$=LEFT$(r$,length%)   ' trim string to length
```

Visual BASIC:

```
CALL ENTER (r$,30,length%,7,status%)
```

Turbo Pascal or Delphi:

```
enter (r,30,len,7,status);
```

C or C++:

```
enter (r,30,&len,7,&status);
```

Using **Initialize**, **Send**, and **Enter** together produces a complete program to obtain data from a device:

QuickBASIC:

```
'Example instrument control program
' $INCLUDE: 'ieeeqb.bi'
'
'-- Initialize the GPIB system
'
CALL INITIALIZE (21,0)
'
'-- Send a command to the instrument
'
CALL SEND (7,"MEASURE",status%)
IF status%<>0 THEN PRINT status% : STOP
'
'-- Read the measured value
'
r$=SPACE$(30)          ' allocate room for data
CALL ENTER (r$,length%,7,status%)
IF status%<>0 THEN PRINT status% : STOP
r$=LEFT$(r$,length%)      ' trim string to length
'
PRINT "Value is ";VAL(r$)

'To run a QuickBASIC program which uses these
'routines, you must load the '488 library file
'when you start QuickBASIC:  C>  QB /L IEEEQB.QLB
```

Visual BASIC:

```
'Example instrument control program
'-- Initialize the GPIB system
'
CALL INITIALIZE (21,0)
'
'-- Send a command to the instrument
'
CALL SEND (7,"MEASURE",status%)
IF status%<>0 THEN PRINT status% : STOP
'
'-- Read the measured value
'
CALL ENTER (r$,30,length%,7,status%)
IF status%<>0 THEN PRINT status% : STOP
'
PRINT "Value is ";VAL(r$)
```

Turbo Pascal or Delphi:

```
PROGRAM example;
{for Turbo Pascal use: ieeepas, for Delphi: ieeedel }
USES ieeepas;
VAR
      status : integer;
      len : word;
      r : string;
BEGIN
{ Initialize the GPIB system }
initialize (21,0);

{ Send a command (take a measurement) }
send (7,"MEASURE",status);
if (status <>0) then halt;

{ Read the measured value }
enter (r,30,len,7,status);

writeln ('Value is ',r);
END.
```

C or C++:

```
#include "ieee-c.h"
main ()
{
      int status,len;
      char r[80];

      /* Initialize the GPIB system */
      initialize (21,0);

      /* Send a command  (take a measurement) */
      send (7,"MEASURE",&status);
      if (status != 0)
            {printf ("%d\n",status); exit(1);}

      /* Read the measured value */
      enter (r,30,&len,7,&status);

      printf ("Value is %s\n",r);
}
```

GPIB devices can provide status information through two mechanisms known as serial polling and parallel polling. There is also a signal, called SRQ (service request), which is used by devices to signal that they require some action.

You can test for service requests with the **Srq** function. This function returns a TRUE value if any device is requesting service. To determine which device is requesting service, you can use the serial poll routine (see **Spoll**, later).

BASIC (QuickBASIC, QBASIC, or Visual BASIC):

```
WaitSRQ:
     IF NOT(srq%) THEN GOTO WaitSRQ
```

Turbo Pascal or Delphi:

```
while (not(srq)) do begin end;
```

C or C++:

```
while (!srq()) ;
```

A serial poll reads the status of a single device.

SPOLL (address,poll,status)

where:

- address is the GPIB address of the device to be polled.
- poll will contain the poll result byte.
- status indicates whether the poll was okay.
  0=okay
  8=timeout (no device responding)

The serial poll status from a device is usually interpreted as a set of 8 bits, each of which may have some device-dependent meaning, such as "measurement complete", "error", or "out of range". The next-to-leftmost bit (hex 40) is reserved to indicate whether the device is requesting service (on the SRQ line).

In the examples that follow, the program tests for service request and stays in a loop calling **Spoll** until the status has one particular bit true.

BASIC (QuickBASIC, QBASIC, or Visual BASIC):

```
WaitStatus:
      CALL SPOLL (7,poll%,status%)
IF (poll% AND 4)=0 THEN GOTO WaitStatus
```

Turbo Pascal or Delphi:

```
repeat
      spoll (7,poll,status);
until ((poll and 4)<>0);
```

C or C++:

```
do {
      spoll (7,&poll,&status);
}     while ((poll & 4) == 0);
```

Parallel polling can also be useful in determining device status. A parallel poll returns a single byte, each bit of which can indicate that a device is requesting service. In this way, the need to separately poll each device can be avoided.

PPOLL (poll)

where:

- poll returns with the poll value (0 to 255).

Ppoll has no status return argument because no timeouts are possible.

Each bit of the poll response value can indicate one or more devices requesting service. Devices may be assigned to certain bits either through configuration inside the device (possibly a fixed bit number for some devices), or through the parallel poll configuration commands (see parallel poll setup under **Transmit**, later).

BASIC (QuickBASIC, QBASIC, or Visual BASIC):

```
CALL PPOLL (poll%)
```

Turbo Pascal or Delphi:

```
ppoll (poll);
```

C or C++:

```
ppoll (&poll);
```

It is often useful to test whether a listener is actually present on the GPIB before beginning to send data to it. At the start of a program, the user can be warned that the device is not responding and may be disconnected, or turned off.

Some models of IEEE-488 interface contain special hardware to allow the presence of a listener to be detected without sending any data.

To check if the hardware supports this feature, call the **GPIBFeature**() routine (described in more detail in the next section). Once you know the feature is supported, you can call the listener present routine.

ListenerPresent (addr)

where:

- addr is the device address you wish to check

returns: 0 if the device is not present, non-zero if it is present.

BASIC example

```
IF (GPIBFeature%(IEEEListener)<>0 AND
(ListenerPresent%(8)=0)) THEN STOP
```

Turbo Pascal / Delphi example:

```
if gpib_feature (IEEEListener) and
not(listener_present(8)) then halt;
```

C or C++:

```
if (gpib_feature(IEEEListener) &&
!listener_present(8)) exit(1);
```

If your hardware does not support this function, you can still detect non-connected devices, because the **Send** routine will return a status of 8 (indicating timeout) immediately, without waiting for the full timeout period, if no device is present.

Testing for the Presence of the GPIB Board

You may find it useful to test whether a board is actually present in the computer before starting to send device commands.

GPIBBoardPresent

returns: 0 if no board is present, non-zero if board is installed

(Note: for backward compatibility, this routine still returns different codes for some different models of interface board, but this is not the correct way to identify board functionality. See the feature routine in the next section).

BASIC:

```
IF (NOT(GPIBBoardPresent%)) THEN ...
```

Turbo Pascal or Delphi:

```
if not(gpib_board_present) then ...
```

C or C++:

```
if (!gpib_board_present()) ...
```

It may be useful to check on the features and settings of the board in your programs.

GPIBFeature (feature)

where

- feature is a number (or named constant) from the following list, indicating which information you wish to retrieve

Feature	number	Description
IEEEListener	0	Does the board support the ListenerPresent function? If not, ListenerPresent will always return true.
IEEE488SD	1	Does the board support the 488SD high-speed protocol?
IEEEDMA	2	Does the board use DMA hardware?
IEEEIOBASE	100	Return the base I/O address
IEEETIMEOUT	200	Return the current timeout setting
IEEEINPUTEOS	201	Return the current input EOS
IEEEOUTPUTEOS1	202	Return the current output EOS1
IEEEOUTPUTEOS2	203	Return the current output EOS2
IEEEBOARDSELECT	204	Return the current board number
IEEEDMACHANNEL	205	Return the current DMA channel

returns: the desired information about the interface board. For yes/no feature inquiries, a zero or non-zero value is returned.

see examples in earlier section on ListenerPresent

Restrictions on the use of Send and Enter

Send and **Enter** are simple routines for transmitting and receiving data. They are carefully designed to handle most instruments but they make some assumptions that may not be valid in your application. Specifically,

- Your board must be the system controller. **Send** and **Enter** assume that your board is the GPIB controller. If there is another controller and your board is being used as a device you should read section 4: Advanced Programming.

- The instrument must accept a line feed or EOI and send a line feed or EOI as a data terminator. **Enter** terminates upon receipt of a line feed or EOI, and **Send** adds a line feed with EOI to the end of the transmitted string. This is what most instruments require, however, if your instrument has other requirements you will need to use **Transmit** and **Receive**.

- The instrument is transmitting character data rather than binary data which could include imbedded linefeeds. We provide **Tarray** and **Rarray** for binary data.

The program TRTEST, provided on the programming and applications disk, is a good program to experiment with if you want to test the use of **Send** and **Enter** with your instruments.

The IEEE-488 standard defines a number of specialized commands in addition to simple data transfers. The **Transmit** command gives you the ability to program any command and exercise a finer level of control over the GPIB than is provided with **Send**.

TRANSMIT (command,status)

where:

- command is a string containing a series of GPIB commands and data. Each item is separated by one or more spaces. A complete list of the available commands is given on the following pages.

- status indicates whether the commands went okay.
  0=okay
  1=illegal command syntax
  2=tried to send data when not a talker
  4=a quoted string or END command was found in a LISTEN or TALK list
  8=timeout or no devices listening
  16=unknown command

The status value is somewhat more complex for **Transmit**. It can be any value from 0 to 31, where each bit indicates a particular type of error. A zero value, as usual, means that the transfer went OK. A value of 8 indicates a timeout; there is nothing wrong with the command, but the instrument is not responding. The other values typically result from typing mistakes in the command string. As an example, if status=24, both 16 and 8 are set, meaning that an unknown command was found in the string and a timeout also occurred.

The examples shown on the next page program two devices to receive data (listen) at the same time and then synchronize their measurements using the Group Execute Trigger command.

The command string is interpreted by the **Transmit** routine as a series of GPIB commands to be carried out. In this case, "UNL" means Unlisten, which disables any listeners that may exist. The sequence "LISTEN 4 7" assign devices at addresses 4 and 7 to be listeners. Finally, "GET" specifies the Group Execute Trigger command.

QuickBASIC, QBASIC, or Visual BASIC:

```
CALL TRANSMIT ("UNL LISTEN 4 7 GET",status%)
```

Turbo Pascal or Delphi:

```
transmit ('UNL LISTEN 4 7 GET',status);
```

C or C++:

```
transmit ("UNL LISTEN 4 7 GET",&status);
```

LISTEN

LISTEN defines one or more listeners. LISTEN should be followed by a series of numbers indicating the GPIB addresses of the devices.

Examples:
"LISTEN 1"
"LISTEN 4 9 30"

TALK

TALK defines a talker. There can only be one talker at a time. If multiple talk commands are given, the last one in the list takes effect.

Examples:
"TALK 3"
"TALK 9"

SEC

SEC defines a secondary address. This command should be followed by a number. SEC is used after the primary address of the device is sent with LISTEN or TALK.

Examples:
"LISTEN 4 SEC 8"
"TALK 8 SEC 9"

UNT

Untalk. Disables the current talker, if any.

UNL

Unlisten. Disables any currently assigned listeners.

MTA

My Talk Address. Assigns your board as the talker.

MLA

My Listen Address. Assigns your board as a listener.

DATA

Indicates that data follows, which should be transmitted to all listening devices. The computer should be assigned as a talker before giving this command. Data may be indicated in two forms: as a string enclosed in single quotes ('), or as numbers from 0 to 255. Quoted strings are sent as characters, just as given in the string. Numbers indicate a byte value to be sent as data. They are useful for sending non-printing characters such as carriage return (13) and line feed (10).

END

Send terminator byte(s) (default is line feed with the EOI signal). This should only be used after the DATA command.

The set of commands shown above will carry out all data transfers. Other commands follow for specialized GPIB control.

"UNL UNT LISTEN 4 MTA DATA 'hello' END"

this command string turns off all listeners and talkers (UNL UNT), then assigns device 4 as a listener, the computer as a talker (MTA), and sends the string 'hello' as data to device 4. Finally, a line feed with EOI is sent (END).

"DATA 'testing' 13 10"

this command assumes that the computer is already a talker, and one or more devices are listening (this could have been set up in a previous call to **Transmit**). The data string 'testing' is sent, followed by a carriage return and a line feed.

"DATA 27 '&k2S'"

this data sequence sends an Escape (ASCII 27), followed by '&k2S'.

"UNL LISTEN 4 8 DATA 'info' 13 10 'second line' 13 10"

this command turns off all listeners, then assigns devices 4 and 8 as listeners, then sends two lines of data to those devices, where each line ends with a return and a line feed.

The command string for a **Transmit** call can be built up out of string and numeric variables in your program.

BASIC

Numeric variables must first be converted to strings with BASIC's STR$ function.

```
VOLTAGE=3.5
CMD$="MTA LISTEN 1 DATA 'V="+STR$(VOLTAGE)+"' 13 10"
CALL TRANSMIT (CMD$,STATUS%)
```

In this example, the variable VOLTAGE is converted to a string and placed within a command string to be used with **Transmit**. The first part of the string makes the computer a talker (MTA), and begins sending data with the quoted string 'V=. The final portion of the string, placed after the converted variable, closes the quoted string and adds a return and line feed. Note: after the line executes, CMD$ will be "MTA LISTEN 1 DATA 'V= 3.5' 13 10".

C or C++

Use the sprintf() routine in the C language to build your command string. For example:

```
sprintf (cmd, "mta listen 1 data 'v=%f' 13 10",
            volts);
transmit (cmd, &status);
```

REN

Remote Enable. This command turns on the remote enable signal. Only the system controller can supply this signal. Remote enable is required by some devices before they will accept commands.

EOI

End-or-Identify. This command operates like the DATA command, but sends the data byte that follows with the EOI signal, to indicate that it is the last byte of a transmission. See examples below.

GTL

Go To Local. Tells the currently listening devices to resume operation from their front panels (as opposed to remote control by the computer). This command also turns off the remote enable signal.

Examples

"MTA LISTEN 4 REN DATA 'hello' 13 10"

this command will send 'hello' followed by return and line feed to the device at address 4. Remote enable is turned on before the data is sent. Note: Remote enable will remain on until it is turned off by calling **Initialize** or using the GTL command.

"DATA 'hello' EOI 10"

this command sends the data string 'hello', followed by a line feed with the EOI signal. ("EOI 10" is equivalent to "END").

"UNL LISTEN 5 8 GTL"

this command makes devices 5 and 8 listen, then tells them to resume front panel operation.

SPE

Serial poll enable. This command is used to obtain a serial poll response from a device. It is used within the SPOLL routine. When a device is addressed to talk and receives the SPE command, it will send its serial poll response instead of normal data.

SPD

Serial poll disable. This command places the polled device back in a normal talker state.

Example

"UNL MLA TALK 5 SPE"

this command sets up the device at address 5 to provide a serial poll response.

PPC

Parallel poll configure. This command tells the currently addressed listener(s) to expect a parallel poll enable command to follow.

A parallel poll enable command is a GPIB command byte between 96 and 111 (sent with the CMD command, see next page). This command is interpreted as a binary byte in the form: 0110SPPP, where S specifies the bit value to be used by the device when it requests service (0 or 1), and PPP specifies a binary value from 0 through 7, indicating the bit number to be used for the response.

For example, a parallel poll enable command of 105 is 01101001 in binary. The final 001 means that bit number 1 (counting from right to left with the rightmost bit as 0) will be used, and the 1 preceeding this indicates that a 1, or TRUE value, will be placed in this bit when the device needs service.

PPD

Parallel poll disable. This command can also follow the PPC command. It disables any parallel poll response for the device being configured.

PPU

Parallel poll unconfigure. This command disables all parallel poll responses on all devices (whether addressed to listen or not).

Note: not all devices implement all of the parallel poll capabilities.

Examples

"PPU"

this command disables all parallel poll responses.

"UNL LISTEN 1 2 PPC PPD"

this command disables the parallel poll responses of devices at GPIB addresses 1 and 2.

"UNL LISTEN 5 PPC CMD 105"

this command enables the parallel poll response of the device at GPIB address 5 to be a 1 value on bit number 1.

DCL

Device Clear. Tells all devices to reset to a predefined state. The action taken by devices upon receiving this command is device dependent.

LLO

Local Lockout. Disables front panel control of devices. This command is usually used in conjunction with REN, to ensure complete control by the computer, and disallow the use of the front panel controls on the devices. Not all devices implement this command.

CMD

Command. This operates in a manner similar to the DATA command, except that the information that follows CMD is sent with the ATN line on, making the bytes GPIB commands. This allows you to send any GPIB command byte, even those that are non-standard. One common use of CMD is to send a parallel poll enable command (see previous page).

GET

Group Execute Trigger. A GPIB command that causes all devices currently addressed to listen to start a device dependent operation (usually a measurement). This command is useful when trying to synchronize operations on multiple devices. Some GPIB devices do not implement this command.

SDC

Selected Device Clear. This command is similar to DCL, but only resets devices currently addressed to listen.

TCT

Take control. This command is used by the controller to allow another device with controller capability to take over control of the GPIB. See the section on Advanced Programming for more details on passing control.

IFC

Interface clear. This command may only be sent by the system controller. It resets the interface state of all devices on the GPIB system. After this command, no devices will be addressed to talk or to listen. If control had been passed to another device, the system controller will regain control of the GPIB. Note: the **Initialize** routine uses this command internally.

The **Transmit** routine can be used to send a device clear. There are two types of device clear: universal clear and selected device clear.

To send a universal clear, use this transmit command string:

"DCL"

To send a selected device clear, address the desired devices to listen and then use an SDC. For example, to clear device 6 use this transmit command string:

"UNL LISTEN 6 SDC"

The **Transmit** routine can be used to send a device trigger command. First address the desired devices to listen (one or more devices) and then use a Group Execute Trigger (GET). For example, to trigger device 9 use this transmit command string:

"UNL LISTEN 9 GET"

Receive can be used to read data from a device. It is similar to **Enter**, but it does not address a talker or establish the PC as a listener on the GPIB. Because of this, it must be used with **Transmit**.

RECEIVE (recv,maxlength,length,status)

where:

- recv is a string variable which will contain the received data. recv must be initialized to a string containing at least as many characters as you wish to receive. **Receive** will terminate reception of data when: 1) the string is full, 2) a line feed is received, or 3) any character is received with the EOI signal. Carriage returns in the incoming data are ignored, and not placed in recv.
- maxlength is a value specifying the maximum number of characters you wish to receive. In BASIC and QuickBASIC, this argument is not present. maxlength can be a number from 0 to 65535 (hex FFFF).
- length will contain the actual number of characters received.
- status indicates whether the transfer went OK.
  0=okay
  2=tried to receive when PC was not a listener
  8=timeout

In DOS BASIC, space is reserved for the incoming data in the same way it was done when **Enter** was used. A string variable is set to spaces with the SPACE$ function. **Receive** returns the received data and the length.

Receive is useful in situations where **Enter** cannot be used. This may occur either when receiving long strings in pieces, by calling **Receive** repeatedly, or when the computer is not the GPIB controller.

Note: if a timeout occurs during **Receive**, it is possible that some data may still have been read. This can occur if the device sends some characters, then pauses for a long time before continuing. In this case, length will be non-zero, and status will be 8. To continue receiving simply call **Receive** again to get the rest of the data.

QuickBASIC or QBASIC:

```
CALL TRANSMIT ("MLA TALK 8",status%)
r$=SPACE$(30)
CALL RECEIVE (r$,length%,status%)
r$=LEFT$(r$,length%)
```

Visual BASIC:

```
CALL TRANSMIT ("MLA TALK 8",status%)
CALL RECEIVE (r$,30,length%,status%)
```

Turbo Pascal or Delphi:

```
transmit ("MLA TALK 8",status);
receive (r,30,len,status);
```

C or C++:

```
transmit ("MLA TALK 8",&status);
receive (r,30,&len,&status);
```

Some instruments can send or accept data in a binary format. The **Send**, **Enter**, and **Receive** routines are not appropriate for binary data because they interpret certain bit patterns as special ASCII characters. **Send** adds a line feed to the data, and **Enter** and **Receive** both terminate if a line feed is received.

The **Tarray** and **Rarray** routines are provided to handle binary data. In addition, these routines are optimized to provide faster data transfer. Neither of these routines does any GPIB addressing for you, so you will need to set up talkers and listeners with **Transmit** before calling them.

Tarray

Tarray allows you to send a long block of binary data from the computer to the current set of listening devices. The EOI signal can optionally be sent along with the last data byte.

TARRAY (data.array,count,eoi,status)

where:

- data.array is the information to be transmitted.
- count is the number of bytes to be transmitted. Note: In BASIC, when you want to send more than 32767 bytes, you will have to assign the value to count% in hex. Example: COUNT%=&HA000
- eoi indicates whether or not to send EOI with the last data byte. 0=NO, 1=YES.
- status indicates whether the transfer went okay.
  0=OK
  2=tried to send when PC was not a talker
  8=timeout

Tarray sends bytes just as they are found in the computer's memory.

QuickBASIC or Visual BASIC:
(note: QBASIC 1.1 does NOT support tarray() and rarray() calls)

```
DIM INFO%(1000)       ' array containing data
... fill the array with data ...
```

```
CALL TRANSMIT ("MTA LISTEN 8",status%)
CALL TARRAY (INFO%(1),2000,1,status%)
```

Turbo Pascal or Delphi:

```
transmit ("MTA LISTEN 8",status);
tarray (info,2000,TRUE,status);
```

C or C++:

```
transmit ("MTA LISTEN 8",&status);
tarray (info,2000,1,&status);
```

The **Rarray** routine is used to receive up to 64K bytes of binary data. It terminates either when the given number of bytes have been received, or when a byte arrives with the EOI signal.

RARRAY (data.array,count,length,status)

where:

- data.array is the array which will contain the received data.
- count is the number of bytes to be received. Note: In BASIC, when you want to receive more than 32767 bytes, you will have to assign the value to count% in hex. Example: COUNT%=&HA000
- length returns the actual number of bytes received.
- status indicates whether the transfer went okay.
  0=okay
  2=tried to receive when PC was not a listener
  8=timeout
  32=successful transfer ended with EOI

Important note: the status value returned by **Rarray** can be non- zero even when the transfer is okay. **Rarray** returns a status of 32 when the transfer was terminated with an EOI signal.

QuickBASIC or Visual BASIC:
(note: QBASIC 1.1 does NOT support tarray() and rarray() calls)

```
DIM INFO%(1000)        ' array containing data
CALL TRANSMIT ("MLA TALK 8",status%)
CALL RARRAY (INFO%(1),2000,length%,status%)
```

Turbo Pascal or Delphi:

```
transmit ("MLA TALK 8",status);
rarray (info,2000,len,status);
```

C or C++:

```
transmit ("MLA TALK 8",&status);
rarray (info,2000,&len,&status);
```

In most cases, you should declare the data array for binary data to be a byte, integer, or real type variable. (Note: in BASIC, byte type variables are not available). The type you choose depends on the data format sent by the instrument.

Sometimes, the data sent by the instrument is not in a format which can readily be used by the PC or by the language you are writing in. This is very common if you are programming in interpreted BASIC, which has only integer and floating point variables (which are stored in a format invented just for BASIC).

One common case is a device which sends 16-bit data, but the bytes are in the reverse order from the way the PC stores them.

Another common case is a device which send 4-byte floating point data, but again the bytes are in reverse order. (Note: in Turbo Pascal, the REAL data type is a 6-byte format specific to Pascal, but the SINGLE data type is an industry-standard 4-byte format).

In these cases, you will have to reformat the data, accessing it as individual bytes and rearranging them so the computer can process them. Here is sample code in BASICA for reversing the bytes in 16-bit data:

```
400 DEF SEG     ' access BASIC's data segment
405 I=0 : J=0                    ' initialize
variables
410 OFS%=VARPTR(info%(1))        ' get memory address
420 FOR I=0 TO count%-2 STEP 2   ' loop through data
430   J=PEEK(OFS+I)              ' swap data bytes
440   POKE OFS+I,PEEK(OFS+I+1)
450   POKE OFS+I+1,J
460 NEXT I
470 DEF SEG=IEEE                 ' 488 code segment
```

You may also want to investigate having the device transmit data in another format, such as ASCII characters, to avoid the data format conversion problem.

Our IEEE-488 boards all include hardware for high speed transfers.

To get the best data rates, use the **Tarray** and **Rarray** routines. The other routines (Send and Enter) are designed for ease of use and mainly for shorter text strings. Tarray and Rarray are optimized to use the high speed hardware features of each board.

Transfer speed is usually limited by the GPIB device. Most instruments have maximum rates of 100K bytes/second or less.
All our boards can transfer data many times faster than this rate.

16-bit ISA bus interface (488EX)

488EX uses 16-bit I/O operations and special hardware to achieve rates of **over 1M bytes/second**. The **Tarray** and **Rarray** routines automatically use this hardware, and there is no need to call any additional routines. However, if you are writing code that may run on either 488EX or one of the other boards, you may want to include a **DmaChannel** call (see below). This call has no effect on 488EX, but will allow the other boards to go faster.

PCI bus interface

PCI488 does not use DMA, so the **DmaChannel** call is ignored for this board. Other hardware methods are used for optimium transfer rate on this board.

Other ISA bus interface models

These boards use direct memory access (DMA), which is built into the computer. PC488 can use DMA channels 1 or 3. 4x488 uses DMA channel 1. PS488 can use DMA channels 0 to 7 in a Micro Channel computer.

Note that other add-in boards may also use DMA, and a channel must be selected that does not conflict with other boards. Channel 1 is usually available.

DMA is initially disabled. To enable the use of DMA, call the **DmaChannel** routine:

DMACHANNEL (channel)

where:

- channel is the DMA channel number used by the interface board. channel **must** match the hardware configuration setting of the board. To disable DMA again, call **DmaChannel** with a channel value of **-1**.

Note: on some hardware models (see above), this call is ignored because the hardware does not use or support DMA transfers. It is OK to call this routine anyway.

QuickBASIC, QBASIC, Visual BASIC:

```
CALL DMACHANNEL(1)
```

Turbo Pascal or Delphi:

```
dmachannel(1);
```

C or C++:

```
dmachannel(1);
```

When an application requires a non-standard hardware setup or non-standard interface settings, you can change these values with the routines described in this section. In most cases, these routines are not needed.

SetPort

SETPORT (board,port)

where:

- board is the interface board number, from 0 to 3. When you have only one IEEE-488 interface board, the board number to use is zero.
- port is the I/O address of the board. The factory default setting is 2B8 hex.

The **SetPort** routine is used when the interface board is set to an I/O address other than the factory default of 2B8 hex. This can occur because of a conflict with other hardware in the computer, or when multiple board interfaces are used.

NOTE: for plug-and-play boards such as PCI bus boards, this call will be ignored, since the I/O port address is known automatically.

NOTE ALSO: even if you do not call SetPort() in your program, your application can still be used on boards set to non-default I/O addresses. See the following section on configuration files.

QuickBASIC, QBASIC, or Visual BASIC:

```
CALL SETPORT (2,&H2A8)
```

Turbo Pascal or Delphi:

```
setport (2,$2A8);
```

C or C++:

```
setport (2,0x2A8);
```

BOARDSELECT (board)

where:

- board is the interface board number, from 0 to 3.

BoardSelect is used only when multiple GPIB interfaces are installed in the computer.

QuickBASIC, QBASIC, or Visual BASIC:

```
CALL BOARDSELECT (2)
```

Turbo Pascal or Delphi:

```
boardselect (2);
```

C or C++:

```
boardselect (2);
```

SETTIMEOUT (msec)

where:

- msec is the new timeout value in milliseconds. The timeout period is the maximum time allowed between input or output bytes before declaring an error. When you are using DMA for high speed transfers, the timeout period applies to the entire transfer, not the time between bytes. msec may be rounded to the nearest multiple of 55 milliseconds.

The default timeout period is 10 seconds (or 10000 msec).

SetTimeout is used if the default timeout period of 10 seconds is not suitable for your application. If you have a very slow device, you may need to lengthen the timeout. If you have a fast device and wish to detect errors in less than 10 seconds, you can shorten the timeout.

QuickBASIC, QBASIC, or Visual BASIC:

```
CALL SETTIMEOUT (3000)
```

Turbo Pascal or Delphi:

```
settimeout (3000);
```

C or C++:

```
settimeout (3000);
```

SETOUTPUTEOS (eos1,eos2)

where:

- eos1 and eos2 are the terminating characters to be sent at the end of the **Send** routine, or when the END command is used in **Transmit**. If eos2 is a null character (0), only one character is sent.

The default output end-of-string is a line feed.

SetOutputEOS is used when you have a device which requires a terminating character other than the default. Some devices require both a return and a line feed. You can also get full control over the data bytes that are sent by using the **Transmit** routine.

QuickBASIC, QBASIC, or Visual BASIC:

```
CALL SETOUTPUTEOS (13,10)
```

Turbo Pascal:

```
setoutputeos (13,10);
```

C:

```
setoutputeos (13,10);
```

SETINPUTEOS (eos)

where:

- eos is the terminating character to be used by the **Enter** and **Receive** routines. If eos = line feed (10), then carriage return characters are removed from the input string.

The default input end-of-string is a line feed.

SetInputEOS is used if you have a device which terminates its transmissions with a character other than line feed. Note that **Enter** and **Receive** also terminate reception when they receive a pre-defined number of characters, or when the GPIB EOI signal is received.

QuickBASIC, QBASIC, or Visual BASIC:

```
CALL SETINPUTEOS (13)
```

Turbo Pascal or Delphi:

```
setinputeos (13);
```

Cor C++:

```
setinputeos (13);
```

For non-plug-and-play interface boards (for example, ISA bus boards with switch settings), you can create a configuration file to tell the driver software about the hardware settings.

The file **CEC488.INI** is used by the software to find the boards. The file format is documented in comments within the file, and you can easily edit this file with any text editor.

The process of finding boards works as follows:

1. First, the **CEC488.INI** file is read, and any boards listed in this file will be used. If a board listed in the file is not actually installed at the given I/O address, that entry is ignored.
2. Next, plug-and-play boards, such as PCI boards, are automatically detected.
3. Finally, if no boards have been found, the default I/O base address of 2B8 is checked to see if a board is present.

In the normal situation of a single IEEE-488 interface board installed, that board will always be board #0 (the default board). If you have more than one IEEE-488 board installed, they will be assigned based on the search order listed above.

The file **CEC488.INI** must be placed in a location that can be found by the software drivers. The location of the file depends on the operating system as follows:

DOS

The file **CEC488.INI** can be placed in the root (C:\) directory, or in the DOS (C:\DOS) directory.

Windows

The file **CEC488.INI** should be placed in the Windows directory.

Windows NT

The file **CEC488.INI must** be placed in C:\WINDOWS, even if that is not the directory in which Windows NT is installed.

How much wood would a woodchuck chuck?
Woodchucks "hibernate in a true torpor for as long as eight months each year", leaving little time for chucking.
- Encyclopedia Britannica

The computer is normally a system controller. It has the privilege of issuing commands to any device at any time and the attendant responsibility of maintaining order and control. In some situations, it may be advantageous to have one computer be controlled by another. For this to happen, the computer must take on new responsibilities and relinquish some of its old privileges.

The most important limitation on any GPIB device which is not a controller is that it cannot send GPIB commands such as talk and listen addresses. This means that the **Send**, **Enter**, **Spoll**, and **Ppoll** routines cannot be used. **Transmit** can be used for data transmission only. **Receive**, **Tarray**, and **Rarray** can all be used.

To become a device on the GPIB, the computer must implement the following major functions:

- Setting its own GPIB address
- Determining its addressing status (talker/listener)
- Sending and/or receiving data
- Requesting service and setting its serial poll response

Setting your board's GPIB address is handled through the **Initialize** routine. When **Initialize** is called with its second argument equal to two, your board will act as a device.

Notes:
PC488 **rev. C or earlier** must have switch S1 position 8 ON to be a device. All other models of hardware handle this function in software - no switch or jumper settings are required.

The tables on the following page show the internal registers of the interface chip on the card. These tables will be referred to during the remainder of this discussion. Only those bits which are important to the discussion in this section are highlighted. More detailed information on these interface chips is available from your local integrated circuit distributor, or NEC sales office.

The NEC 7210 chip is accessed with output and input instructions. For example:

BASIC:

```
OUT &H2BA,&H40        ' enable SRQ interrupt
```

Turbo Pascal:

```
Port[$2BA] := $40;
```

C:

```
outp (0x2BA,0x40);
```

Input:

Addr. (hex)	Name	Bit assignment							
2B8	data	D7	D6	D5	D4	D3	D2	D1	D0
2B9	status 1	---	---	GET	END	DEC	ERR	DO	DI
2BA	status 2	---	SRQ	---	---	CO	---	---	ADC
2BB	spoll status	S8	PEND	S6	S5	S4	S3	S2	S1
2BC	address stat	CIC	---	---	---	---	LA	TA	---
2BD	command pass	---	---	---	---	---	---	---	---
2BE	address 0	---	---	---	---	---	---	---	---
2BF	address 1	---	---	---	---	---	---	---	---

Output:

2B8	data	D7	D6	D5	D4	D3	D2	D1	D0
2B9	mask 1	---	---	GET	END	DEC	---	DO	DI
2BA	mask 2	---	SRQ	---	---	CO	---	---	ADC
2BB	spoll resp.	S8	RSV	S6	S5	S4	S3	S2	S1
2BC	address mode	---	---	---	---	---	---	---	---
2BD	aux. command	D7	D6	D5	D4	D3	D2	D1	D0
2BE	address 0/1	---	---	---	---	---	---	---	---
2BF	end of string	---	---	---	---	---	---	---	---

In a typical application, once your board is initialized as a device, it will wait in an idle loop until it is addressed to talk or to listen. The "TA" and "LA" bits in the address status registers indicate the talk/listen state of the interface.

BASIC:

```
'--- wait until addressed to talk or listen
Lp:
   IF (INP(&H2BC) AND 2)<>0 THEN GOTO Tk
   IF (INP(&H2BC) AND 4)<>0 THEN GOTO Ln
   GOTO Lp
Tk: '--- TALK
...
Ln: '--- LISTEN
...
```

Turbo Pascal:

```
while true do begin
  if (Port[$2BC] and 2)<>0 then begin
      { talk }
  end;
  if (Port[$2BC] and 4)<>0 then begin
      { listen }
  end;
end;
```

C:

```
while (1) {
      if (inp(0x2BC) & 2)
            ... /* talk */
      if (inp(0x2BC) & 4)
            ... /* listen */
}
```

Once the computer has been addressed to talk, any of the data transmission routines can be called. If it has been addressed to listen, **Receive** or **Rarray** should be called.

The following example expands the address status testing shown earlier into a complete program.

In this example, the device will send a data string after it has received the string "MEASURE". The variable COMMAND indicates whether or not the MEASURE string has been received. COMMAND is used to insure that **Receive** is only called once when the computer becomes a listener, and data is transmitted only once per receipt of "MEASURE".

This example is shown only in BASIC. The more powerful program shown next will be given in all the popular languages.

```
5 '--- Initialize the interface ---
10 DEF SEG=0
12 IEEE=PEEK(&H182)+256*PEEK(&H183)
15 IF IEEE=0 THEN STOP
14 DEF SEG=IEEE
20 INIT=0 : TRANSMIT=3 : RECEIVE=6   ' offsets
30 MY.ADDRESS%=2 : DEVICE%=2
40 CALL INIT (MY.ADDRESS%,DEVICE%)   ' initialize
47 COMMAND=0
50 '--- wait to be addressed ---
55 '
60 IF (INP(&H2BC) AND 2)<>0 THEN GOSUB 1000
70 IF (INP(&H2BC) AND 4)<>0 THEN GOSUB 2000
80 GOTO 50
1000 '--- TALK ---
1002 '
1005 IF COMMAND=0 THEN RETURN   ' no cmd yet
1010 T$="DATA 'data from the PC' 13 END'"
1020 CALL TRANSMIT (T$,STATUS%)
1025 COMMAND=0
1030 RETURN
2000 '--- LISTEN ---
2005 '
2007 IF COMMAND=1 THEN RETURN   ' done already
2010 R$=SPACE$(40)
2020 CALL RECEIVE (R$,LENGTH%,STATUS%)
2030 R$=LEFT$(R$,LENGTH%)
2040 IF R$="MEASURE" THEN COMMAND=1
2050 RETURN
```

If you wish to have more detailed control over transmitting and receiving bytes of data, you can directly access the interface chip for these operations as well. The DO, DI, and CO bits on the NEC 7210 indicate readiness to send or receive.

This approach can be useful if you want to insure that the computer can always respond to transmitted data, even after receiving one command. In the previous example, the computer receives a single data string, then insists on transmitting before it will receive again. This limits the ability of the controller to send multiple device commands. You can look at this as a switch that is set to either transmit or receive. If your application requires that transmitting and receiving can occur in any sequence, you will need more detailed control.

Whenever the DI bit is set, a data byte has been received. The program should read this byte from the data register.

When the DO bit is set, the interface is ready for a data byte to be transmitted. This is done by writing to the data register. The CO bit indicates that the NEC 7210 is ready to transmit a GPIB command. This bit is not used when the computer is acting as a device.

The ERR bit is set if you have written a byte to the data register which was not sent. This is because the controller took over the GPIB. You can handle this condition by retransmitting the last byte.

The END bit may be set at the same time the DI bit is set. The END bit indicates that the character just received was accompanied by the EOI signal, indicating the end of a data block.

Important note: reading either the status 1 or the status 2 register from the interface chip automatically clears the register. For this reason, it is important that you do not mix the method of reading the status registers directly with calling **Transmit** and **Receive**. If, for example, you read the status to determine that DI was set (and data had arrived), then called the **Receive** routine, **Receive** would wait forever for the first DI to occur (you have already read it and cleared it).

The examples which follow show a device which can receive or transmit in any sequence.

BASICA or GWBASIC:

```
 10 DEF SEG=0
12 IEEE=PEEK(&H182)+256*PEEK(&H183)
14 IE IEEE=0 THEN PRINT "BASIC488 not loaded" : STOP
16 DEF SEG=IEEE
20 INIT=0
30 MY.ADDRESS%=3 : DEVICE%=2
40 CALL INIT (MY.ADDRESS%,DEVICE%)  ' initialize
50 '
60 OUTBUF$=""             ' reset data buffers
65 LASTBYTE$=""
70 INBUF$=""
75 STATUS=0              ' reset status
80 '
90 '--- Main processing loop ---
100 GOSUB 1000
110 GOSUB 2000
120 IF INBUF$<>"" THEN GOSUB 3000
130 GOTO 100
140 '
1000 '--- TALK ---
1010 GOSUB 4000            ' get status
1015 IF (STATUS AND 4)=0 THEN 1020
1017   OUTBUF$=LASTBYTE$+OUTBUF$
1018   STATUS=STATUS AND &HF7
1020 IF OUTBUF$="" THEN RETURN    ' nothing to say?
1030 IF (STATUS AND 2)=0 THEN RETURN ' not ready?
1035 STATUS=STATUS AND &HFD     ' clear status bit
1040 OUT &H2B8,ASC(OUTBUF$)     ' send one character
1045 LASTBYTE$=LEFT$(OUTBUF$,1)
1050 OUTBUF$=RIGHT$(OUTBUF$,LEN(OUTBUF$)-1) ' delete
1060 RETURN
1065 '
      (continued)
```

```
 2000 '--- LISTEN ---
2010 IF LEN(INBUF$)=255 THEN RETURN ' no room in
buffer?
2020 GOSUB 4000           ' get status
2030 IF (STATUS AND 1)=0 THEN RETURN ' no byte
received?
2035 STATUS=STATUS AND &HFE     ' clear status bit
2040 INBUF$=INBUF$+CHR$(INP(&H2B8)) ' read one
character
2050 RETURN
2055 '
3000 '--- process input data ---
3010 P=INSTR(INBUF$,CHR$(10))    ' look for line feed
3020 IF P=0 THEN RETURN       ' none found?
3030 CMD$=LEFT$(INBUF$,P)      ' extract command
3040 INBUF$=RIGHT$(INBUF$,LEN(INBUF$)-P) ' delete
3050 '
3055 ' handle all valid commands, producing output
3057 '
3060 IF LEFT$(CMD$,7)="MEASURE" THEN
              OUTBUF$=OUTBUF$+"VALUE=3.5"+CHR$(10)
3070 RETURN
3075 '
4000 '--- read status register ---
4010 STATUS=STATUS OR INP(&H2B9)
4020 RETURN
```

Explanation of the previous example:

The variable STATUS is used to keep track of the bits read in from the status 1 register in the NEC 7210. Every time this register is read in, it is ORed with the value in STATUS. In this way, if DI is set, even though we're looking for DO at the moment, it will be remembered.

The variable INBUF$ is used to hold all incoming data. At any time, if the computer is put in a listening state and bytes arrive, they will be added to INBUF$. The computer processes the commands in INBUF$ in the subroutine at line 3000.

The variable OUTBUF$ is used to hold all outgoing data. Any time the computer is addressed to talk and OUTBUF$ contains data, the computer will attempt to send that data.

Lines 10-40: standard initialization as a device at address 3

Lines 60-75: the output and input buffer variables are reset to contain no characters. STATUS is reset.

Lines 100-130: the computer continually checks to see if it should listen, talk, or process commands. Any time INBUF$ contains data, commands are processed.

Lines 1000-1060: Talking. If the ERR bit is set, the last byte is put back in the output buffer (OUTBUF$). If OUTBUF$ contains no data, nothing needs to be done. STATUS is updated. If the DO bit is not set, nothing can be transmitted now, so the subroutine returns. If DO is set, the program clears it and outputs one character.

Lines 2000-2050: Listening. Similar to talking, above.

Lines 3000-3070: Processing commands. All commands for this example device end with a line feed. If no line feed exists in the input buffer, nothing is ready to process yet. If one does exist, the command string up to that line feed is extracted and check against the valid command strings. In this example, only one command is implemented: "MEASURE". This command causes "VALUE=3.5" and a line feed to be added to the output buffer.

You can use this example as a basis for building your own "computer as a device" programs. Simply modify the commands which are processed in subroutine 3000 as needed for your application.

QuickBASIC:

```
' $INCLUDE: 'IEEEQB.BI'
CALL INITIALIZE (3,2)
OUTBUF$=""             ' reset data buffers
LASTBYTE$=""
INBUF$=""
STATUS=0              ' reset status
'--- Main processing loop ---
Loop:
      CALL Talk
      CALL Listen
      IF INBUF$<>"" THEN CALL Process
      GOTO Loop

SUB Talk
  SHARED OUTBUF$,LASTBYTE$,STATUS
  CALL GetStatus                ' get status
  IF (STATUS AND 4)=1 THEN
     OUTBUF$=LASTBYTE$+OUTBUF$
     STATUS = STATUS AND &HF7
  END IF
  IF OUTBUF$="" THEN EXIT SUB  ' nothing to say?
  IF (STATUS AND 2)=0 THEN EXIT SUB ' not ready
  STATUS=STATUS AND &HFD     ' clear status bit
  OUT &H2B8,ASC(OUTBUF$)     ' send one character
  LASTBYTE$=LEFT$(OUTBUF$,1)
  OUTBUF$=RIGHT$(OUTBUF$,LEN(OUTBUF$)-1) ' delete
END SUB
```

(continued)

```
SUB Listen
  SHARED INBUF$,STATUS
  IF LEN(INBUF$)=255 THEN EXIT SUB ' no room
  CALL GetStatus       ' get status
  IF (STATUS AND 1)=0 THEN EXIT SUB ' no byte?
  STATUS=STATUS AND &HFE     ' clear status bit
  INBUF$=INBUF$+CHR$(INP(&H2B8)) ' read one character
END SUB
SUB Process
  SHARED INBUF$,OUTBUF$
  P=INSTR(INBUF$,CHR$(10))    ' look for line feed
  IF P=0 THEN EXIT SUB         ' none found?
  CMD$=LEFT$(INBUF$,P)         ' extract command
  INBUF$=RIGHT$(INBUF$,LEN(INBUF$)-P) ' delete
  ' handle all valid commands, producing output
  IF LEFT$(CMD$,7)="MEASURE" THEN
       OUTBUF$=OUTBUF$+"VALUE=3.5"+CHR$(10)
END SUB
SUB GetStatus
  SHARED STATUS
  STATUS=STATUS OR INP(&H2B9)
END SUB
```

Turbo Pascal:

```
PROGRAM device;
USES ieeepas;
VAR
  status : integer;
  outbuf, inbuf : string;
  lastbyte : char;
PROCEDURE Talk;
BEGIN
  status := status or port[$2b9];
  if (status and 4)<>0 then begin
    outbuf := ' '+outbuf;
    outbuf[1] := lastbyte;
    status := status and $F7;
  end;
  if outbuf<>'' then begin
    if (status and 2)<>0 then begin
      status := status and $FD;
      port[$2b8] := byte(outbuf[1]);
      lastbyte := outbuf[1];
      outbuf := copy(outbuf,2,length(outbuf)-1);
    end;
  end;
END;
PROCEDURE Listen;
BEGIN
  if length(inbuf)<80 then begin
    status := status or port[$2b9];
    if (status and 1)<>0 then begin
      status := status and $FE;
      inbuf := inbuf+' ';
      inbuf[length(inbuf)] := chr(port[$2b8]);
    end;
  end;
END;
```

(continued)

```
PROCEDURE Process;
VAR
      i : integer;
      cmd : string;
BEGIN
  i := pos(#10,inbuf);
  if i<>0 then begin
    cmd := copy(inbuf,1,i);
    inbuf := copy(inbuf,i+1,length(inbuf)-i);
    { handle commands here }
    if (copy(inbuf,1,7)='MEASURE') then
            outbuf := outbuf+'VALUE=3.5'#10
  end;
END;

BEGIN
      initialize (3,2);
      outbuf := '';
      inbuf := '';
      status := 0;
      while true do begin
            Talk;
            Listen;
            if inbuf<>'' then Process;
      end;
END.
```

C:

```
#include <ieee-c.h>
int status;
char inbuf[256],outbuf[256],lastbyte;
main () {
   initialize (3,2);
   status = inbuf[0] = outbuf[0] = 0;
   while (1) {
      talk();
      listen();
      if (inbuf[0]) process();   }
}
talk() {
      status |= inp(0x2b9);
      if (status & 4) {
         memmove (outbuf+1,outbuf,255);
         outbuf[0] = lastbyte;
         status &= 0xF7;  }
      if (!outbuf[0]) return(0);
      if (!(status & 2)) return(0);
      status &= 0xFD;
      outp (0x2b8,outbuf[0]);
      lastbyte = outbuf[0];
      strcpy (outbuf,outbuf+1);
}
listen() {
      if (strlen(inbuf)==255) return(0);
      status |= inp(0x2b9);
      if (!(status & 1)) return(0);
      status &= 0xFE;
      inbuf[strlen(inbuf)+1] = '\0';
      inbuf[strlen(inbuf)] = inp(0x2b8);
}
process() {
   char *p,*strchr(),cmd[80];
   p = strchr(inbuf,'\n');
   if (!p) return(0);
   *p = '\0'; strcpy (cmd,inbuf);
   strcpy (inbuf,p+1);
   if (!strncmp(cmd,"MEASURE",7))  strcat
(outbuf,"VALUE=3.5\n");
}
```

Requesting service

A GPIB device can request service from the controller at any time by asserting the SRQ interface line. Once the controller recognizes the service request, it will usually conduct a serial poll to determine the device status.

You can request service and set the response your computer will give to a serial poll at the same time by writing to the "spoll resp." register. The bit labeled RSV must be a one to cause a service request. The other bits can be any desired value. This byte will be sent as a response to a serial poll.

```
OUT &H2BB,&H41    ' set spoll response and
                  ' request service
```

If, after requesting service, you wish to know whether the controller has done a serial poll yet, you can check the PEND bit in the NEC 7210. The PEND bit is a one after you request service, and becomes zero when a serial poll occurs.

```
105 ' wait until polled ---
110 IF (INP(&H2BB) AND &H40)<>0 THEN 110
```

Note: the SRQ bit is also shown in the register tables. This bit is used only in the controller. It indicates that one or more devices are requesting service.

Other device status bits

Some other status bits are defined in the register tables given earlier. These are the GET, DEC, ADC, and CIC bits.

The **GET** bit is set whenever a Group Execute Trigger is received by the device. This command is often used to start a device dependent operation such as a measurement.

The **DEC** bit is set whenever a Device Clear or Selected Device Clear command is received by the device. This command is often used to cause the device to re-initialize or re-calibrate itself.

The **ADC** bit is set when the addressing status of the device changes (the device becomes a talker or listener or stops being one).

The **CIC** bit is set when the interface board is currently controller-in-charge.

Note that all the bits in the status registers can be used to cause interrupts (see the discussion of Interrupt Processing, later in the manual). To allow a given status bit to cause an interrupt, the equivalent mask bit must be set to a one. For example, to allow DI or DO to cause interrupts on the NEC 7210:

```
300 OUT &H2B9,3      ' unmask DO & DI interrupts
```

The IEEE-488 standard allows a GPIB controller to pass control to another device which has controller capability. After passing control, the computer acts as a GPIB device until control is passed back again.

To pass control, the computer must address a device to talk, then send the Take Control command. When the attention line (ATN) goes low at the end of the **Transmit** call, the new controller takes over.

```
90 TRANSMIT=3
100 CMD$="TALK 4 TCT"       ' give device 4 control
110 CALL TRANSMIT (CMD$,STATUS%)
120 IF STATUS%<>0 THEN STOP    ' check status
```

There is no standard method of asking for control to be returned. It is up to the programmer of the system to provide a means for deciding when to pass control.

Interrupt Processing

Interrupts allow your board to request the attention of your program as needed, in a way that you define.

The most common use of interrupts is in handling service requests (SRQ's) from devices on the GPIB system. Instead of asking every device periodically whether it needs attention, an "interrupt service routine" may be set up which will be executed whenever the correct event(s) occur. After the interrupt processing is complete, control will return to your main program at the point where it was interrupted. Interrupts can also be defined to occur on many other conditions, such as a change in the addressing status (talker/listener) of your board. A description of the interrupt conditions available is given under "The Computer as a GPIB Device", earlier in this manual.

Calling 488 routines within an interrupt

You may use the IEEE-488 routines inside your interrupt service routine. However, some precautions must be taken if you are also calling the '488 routines in the main program.

If a '488 routine is being executed when the interrupt occurs, and the interrupt routine calls a '488 routine, information will be lost. The '488 routines are not designed to be "re-entrant", which means they must not be called again while one of the routines is executing.

To avoid interference between '488 routines in the interrupt procedure and the main program, you should disable IEEE-488 interrupts whenever you call '488 routines in the main program. For example, if you have enabled SRQ interrupts (by outputting a 40 hex to port 2BA), and you want to call the Send routine:

```
500 OUT &H2BA,0          ' disable SRQ intr.
510 CALL SEND (addr%,s$,status%)
520 OUT &H2BA,&H40       ' enable SRQ intr.
```

This example is shown in BASIC, but the same idea applies in any language.

Note that this is not needed if you don't call '488 routines in both the interrupt procedure and the main procedure, or if you can guarantee that the interrupt will never occur while a '488 routine is executing.

Setting up an interrupt service routine

The following steps are required to provide interrupt handling in your programs:

- The board must have the appropriate hardware interrupt jumper installed. Interrupts 2 through 7 may be chosen (see the Hardware Configuration section of the manual (section J) for the jumper locations). Note that other hardware on the PC may use interrupts, and that the board must not conflict with these. Your board comes with interrupts disabled. Interrupt 3 should be OK if you don't have two serial (COM:) ports.
- The board must be initialized, using the INITIALIZE routine.
- The address of the interrupt service routine must be placed into the correct "interrupt vector" location in memory. A double-word address (offset followed by segment) is used. The vector can be set using memory "poke" operations, or through the DOS system call "set vector" (function 25H). Note that the software interrupt numbers used with the DOS call would be 0AH through 0FH. The following list gives the vector locations:

Interrupt #	Vector address (hex) segment:offset
2	0000:0028
3	0000:002C
4	0000:0030
5	0000:0034
6	0000:0038
7	0000:003C

- The interrupt conditions must be defined by setting the interrupt masks in your board's hardware. Again, this is described in the "computer as a gpib device" section of this manual.
- The hardware interrupt line must be "enabled" inside the PC. This is done by clearing a single bit in the interrupt controller chip. For example, if you are using interrupt number 3, you should INPUT a byte from port 21H (say that you get 0B8H), then reset bit number 3 (counting from 0 as the rightmost bit, to give 0B0H), then OUTPUT the byte back to port 21H.

At this point, whenever the defined conditions occur, the interrupt service routine will be executed. Note that the service routine must save all the processor's registers so that it will not mess up the main program's execution. The service routine must end with the following sequence, shown in assembly language:

```
;... pop all registers except AX
CLI         ; interrupts OFF
MOV AL,20H   ; tell PC interrupt is complete
OUT 20H,AL
POP AX       ; restore AX register
IRET        ; return from interrupt
```

Note that it can be dangerous to leave interrupts enabled when your program has terminated. If any more interrupts occur, the PC will jump to the location in memory specified by the interrupt vector, which may no longer be a valid service routine, thus crashing the PC.

Handling interrupts in BASIC

The following pages give examples of handling the SRQ interrupt in various programming languages. If you would like to handle interrupts in DOS BASIC (but not Visual BASIC for Windows), order our "BASIC Interrupt Software" product.

An example program is given below. This example continually prints the "poll_status" variable until a key is hit on the keyboard. The variable will be updated by doing a serial poll whenever a service request interrupt occurs.

Note: this code requires Turbo Pascal version 4.0 or later.

Note: you CAN call the interface routines inside the interrupt service routine. However, there are some limitations. Turbo Pascal does not allow DOS functions (including any type of file or screen I/O) within interrupt routines.

```
PROGRAM SRQtest (input,output);
USES ieeepas,dos,crt;
VAR
            status : integer;
            poll_status : byte;
PROCEDURE int_service; interrupt;
BEGIN
  spoll (1,poll_status,status);
  { signal interrupt processing complete }
  port[$20] := $20;
END;
{------- Main routine -------}
BEGIN
      initialize (21,0);
      { set interrupt vector }
      SetIntVec ($B,@int_service);
      { enable SRQ }
      port[$2BA] := $40;
      { enable interrupt in PC }
      port[$21] := port[$21] and $F7;
      { wait ... and keep printing poll_status byte }
      poll_status := 0;
      while (not(keypressed)) do
      begin
            writeln (poll_status);
            delay (300);
      end;
      port[$21] := port[$21] or 8;
END.
```

Multiple interfaces can be used in the same computer. This can be useful when more than 14 devices must be controlled.

When more than one board is installed in a computer, they must not use the same memory address, I/O address, interrupt channel, or DMA channel. See the Hardware Configuration appendix for details on setting these addresses and channels.

The routines **SetPort** and **BoardSelect** are used when you have multiple interfaces. First, call **SetPort** to tell the software the I/O address for each interface board. Then, use **BoardSelect** whenever you wish to access a particular board.

For example, in QuickBASIC:

```
CALL SetPort (0,&H2B8)
CALL SetPort (1,&H2A8)

CALL BoardSelect(0)
CALL Initialize (21,0)
CALL Send (2,"RESET",status%)

CALL BoardSelect(1)
CALL SetTimeout (3000)
CALL Initialize (21,0)
CALL Send (4,"INIT",status%)
```

After wisdom comes wit.
- Evan Esar

These programming examples illustrate most typical applications. The examples are designed to be useful tools and should provide you with a base to begin your own programming.

Very little code is required to program your board. The code required typically represents a small percentage of the lines in a program since most program lines involve manipulating the received data, writing or reading from files, and displaying the results.

An approach that you may find useful in writing your applications is to use the transmit-receive test routine (TRTEST.EXE) provided on the applications disk. TRTEST allows you to experiment with any combination of bus commands and device commands.

```
10  '-----------------------------------------------
20  ' Example 1: use of SEND & ENTER to communicate
30  ' with an instrument (Keithley 195 meter)
40  '-----------------------------------------------
42  DEF SEG=0      ' find IEEE code memory location
43  IEEE=PEEK(&H182)+256*PEEK(&H183)
44  IF IEEE=0 THEN PRINT "BASIC488 missing" : STOP
45  DEF SEG=IEEE
60  INITIALIZE=0           ' offsets for subroutines
70  SEND=9 : ENTER=21
80  '
90  MY.ADDRESS%=21  ' make PC a controller
100 LEVEL%=0
110 CALL INITIALIZE(MY.ADDRESS%,LEVEL%)
120 '
130 ADDRESS%=16         ' GPIB address of device
140 S$="F0R0X"          ' device command to set mode
150 CALL SEND(ADDRESS%,S$,STATUS%)
160 IF STATUS%<>0 THEN STOP     ' test for errors
170 '
180 R$=SPACE$(80)      ' set up room to receive data
190 CALL ENTER(R$,LENGTH%,ADDRESS%,STATUS%)
200 IF STATUS%<>0 THEN STOP
210 '
220 PRINT "Data received='";LEFT$(R$,LENGTH%);"'"
230 END
```

```
10 '---------------------------------
20 ' Example 2: testing for a service request (SRQ)
30 '---------------------------------
42  DEF SEG=0      ' find IEEE code memory location
43  IEEE=PEEK(&H182)+256*PEEK(&H183)
44  IF IEEE=0 THEN PRINT "BASIC488 missing" : STOP
45  DEF SEG=IEEE
50 INITIALIZE=0 : SEND=9 : ENTER=21 : SRQTEST=63
60 MY.ADDRESS%=21 : LEVEL%=0
70 CALL INITIALIZE(MY.ADDRESS%,LEVEL%)
80 ADDRESS%=4         ' instrument address is 4
90 S$="MEASURE"  ' tell device to take a measurement
100 CALL SEND(ADDRESS%,S$,STATUS%)
110 IF STATUS%<>0 THEN STOP
115 '--- now, wait for SRQ status bit ---
124 '
125 CALL SRQTEST(s%)
127 IF s%=0 THEN 125
130 '
131 ' -- SRQ has occurred, now read the result
140 R$=SPACE$(80)
150 CALL ENTER(R$,LENGTH%,ADDRESS%,STATUS%)
160 IF STATUS%<>0 THEN STOP
170 PRINT "Data received='";LEFT$(R$,LENGTH%);"'"
180 END
```

```
10 '-----------------------
20 ' Example 3: acquiring data for use with another
25 ' program, such as Lotus 1-2-3(tm).
30 '-----------------------
42  DEF SEG=0        ' find IEEE code memory location
43  IEEE=PEEK(&H182)+256*PEEK(&H183)
44  IF IEEE=0 THEN PRINT "BASIC488 missing" : STOP
45  DEF SEG=IEEE
50 INITIALIZE=0 : SEND=9 : ENTER=21
60 MY.ADDRESS%=21 : LEVEL%=0
70 CALL INITIALIZE(MY.ADDRESS%,LEVEL%)
80 ADDRESS%=16     ' the instrument is at address 16
85 '
90 OPEN "DATAFILE.PRN" FOR OUTPUT AS #1 ' open a file
95 '
100'---- Loop to acquire 100 values ---
110 FOR I=1 TO 100
120   R$=SPACE$(80)
130   CALL ENTER(R$,LENGTH%,ADDRESS%,STATUS%)
140   IF STATUS%<>0 THEN STOP
150   PRINT #1,LEFT$(R$,LENGTH%)
160 NEXT I
170 CLOSE
180 '
181 ' The data is now in the disk file.
182 ' For Lotus 1-2-3, use the /File Import Numbers
183 ' command to read the data.
184 ' For a database manager, use the appropriate
185 ' import ASCII data command.
190 END
```

```
10  '-----------------------------------------------
20  ' Example 4: use of TRANSMIT
30  '-----------------------------------------------
42  DEF SEG=0       ' find IEEE code memory location
43  IEEE=PEEK(&H182)+256*PEEK(&H183)
44  IF IEEE=0 THEN PRINT "BASIC488 missing" : STOP
45  DEF SEG=IEEE
50  INITIALIZE=0 : TRANSMIT=3
60  MY.ADDRESS%=21 : LEVEL%=0
70  CALL INITIALIZE(MY.ADDRESS%,LEVEL%)
80  '
90  ' Now, send a trigger to three devices
100 ' at 2,4, and 5, using the LISTEN and GET (group
110 ' execute trigger) commands.
120 '
130 CMD$="REN UNL LISTEN 2 4 5 GET"
140 CALL TRANSMIT(CMD$,STATUS%)
150 IF STATUS%<>0 THEN STOP
160 END
```

```
10 '-------------------------------------------
20 ' Example 5: receiving binary array data
30 ' (from a Tektronix 7612D digitizer)
40 '-------------------------------------------
42  DEF SEG=0     ' find IEEE code memory location
43  IEEE=PEEK(&H182)+256*PEEK(&H183)
44  IF IEEE=0 THEN PRINT "BASIC488 missing" : STOP
45  DEF SEG=IEEE
60  INIT=0 : ADDR%=21 : LEVEL%=0
70  CALL INIT(ADDR%,LEVEL%)
80  TRANSMIT=3 : RARRAY=203 : RECEIVE=6
90  '
100 SDC$="REN LISTEN 1 SEC 0 SDC"  ' device clear
110 CALL TRANSMIT(SDC$,STATUS%)
120 '
130 CMD$="MTA DATA 'REP 0,A' END"  ' Start digitizing
140 CALL TRANSMIT(CMD$,STATUS%)
150 '
160 DIM DATA%(1024)              ' array for data
170 CMD$="MLA TALK 1 SEC 0"      ' Set device to talk
180 CALL TRANSMIT(CMD$,STATUS%)
190 '
200 COUNT%=3 : LENGTH%=0 : S%=-1   ' read 3 byte
205 OFS%=VARPTR(DATA%(1))
210 CALL RARRAY(S%,OFS%,COUNT%,LENGTH%,STATUS%)
220 '
230 COUNT%=2048 : LENGTH%=0 : S%=-1  ' read 2048 bytes
235 OFS%=VARPTR(DATA%(1))
240 CALL RARRAY(S%,OFS%,COUNT%,LENGTH%,STATUS%)
242 ' Note: the data is sent in a non-PC binary format
250 '
260 CALL TRANSMIT(SDC$,STATUS%)   ' clear device
270 END
```

```
PROGRAM example1;
USES ieeepas;
{
      Example 1: use of SEND & ENTER to communicate
      with an instrument (Keithley 195 meter)
}
CONST k195 = 16;  { GPIB address of the instrument }
VAR
      status : integer;
      l : word;
      r : string;
BEGIN
  initialize (21,0);   { make PC controller,addr. 21 }
  send (k195,'F0R0X',status);   { device command }
  enter (r,80,l,k195,status);   { read a voltage }
  writeln ('Data received=',r);
END.
```

```
PROGRAM example2;
USES ieeepas;
{
      Example 2: testing for a service request (SRQ)
}
VAR
      status : integer;
      l : word;
      r : string;
BEGIN
  initialize (21,0);
  send (4,'MEASURE',status);     { start measurement }
  while (not(srq)) do begin end;  { wait for SRQ }
  enter (r,80,l,4,status);        { read results}
END.
```

```
PROGRAM example3;
USES ieeepas;
{
```

```
  Example 3: acquiring data for use with another
  program, such as Lotus 1-2-3(tm).
}
VAR
      datafile : TEXT;
      l : word;
      r : string;
      i : integer;
BEGIN
  initialize (21,0);
  assign (datafile,'DATAFILE.PRN'); { open file }
  rewrite (datafile);
  for i := 1 to 100 do
  begin
     enter (r,80,1,16,status);     { read a value }
     writeln (datafile,r);      { store in the file }
  end;
  close (datafile);
{ The data is now in the disk file.
 For Lotus 1-2-3, use the /File Import Numbers
 command to read the data.
 For a database manager, use the appropriate
 import ASCII data command.
}
END.
```

```
PROGRAM example4;
USES ieeepas;
{
      Example 4: use of TRANSMIT
}
VAR
      status : integer;
BEGIN
      initialize (21,0);
{
 Now, send a trigger to three devices at addresses
 2,4, and 5, using the LISTEN and GET (group execute
 trigger) commands.
}
      transmit ('REN UNL LISTEN 2 4 5 GET',status);
 END.
```

```
PROGRAM example5;
USES ieeepas;
{
      Example 5: receiving binary array data
      (from a Tek 7612D digitizer)
}
CONST
      sdc = 'REN LISTEN 1 SEC 0 SDC';
VAR
      status : integer;
      l : word;
      data : array [1..1024] of integer;
BEGIN
  initialize (21,0);
  transmit (sdc,status);   { selected device clear }

  { start digitizing }
  transmit ('MTA DATA ''REP 0,A'' END',status);

  { set device to talk }
  transmit ('MLA TALK 1 SEC 0',status);

  { read 3 bytes of header info }
  rarray (data,3,l,status);

  { read 2048 bytes of waveform data }
  rarray (data,2048,l,status);
  { note: the device sends data in a non-PC format }

  { clear device to stop it }
  transmit (sdc,status);
END.
```

```
/*
Example 1: use of SEND & ENTER to communicate
 with an instrument (Keithley 195 meter)
*/
#include <ieee-c.h>

#define K195 16

main ()
{
  int status,l;
  char r[80];

  initialize (21,0);   /* make PC controller */
  send (K195,"F0R0X",&status);    /* device command */
  enter (r,80,&l,K195,&status);   /* read a voltage */
  printf ("Data received=%s\n",r);
}
```

```
/*
      Example 2: testing for a service request (SRQ)
*/
#include <ieee-c.h>

main ()
{
  int status,l;
  char r[80];

  initialize (21,0);
  send (4,"MEASURE",&status); /* start measurement */
  while (not(srq())) ;        /* wait for SRQ */
  enter (r,80,&l,4,&status);  /* read results */
}
```

```
/*
Example 3: acquiring data for use with another
```

```
      program, such as Lotus 1-2-3(tm).
*/
#include <ieee-c.h>
#include <stdio.h>

main ()
{
      FILE *datafile;
      int l,i;
      char r[80];

      initialize (21,0);
      datafile = fopen ("DATAFILE.PRN","w");
      for (i=0;id;i++)
      {
         enter (r,80,&l,16,&status);     /* read */
         fprintf (datafile,"%s\n",r);    /* store */
      }
      fclose (datafile);
/*
 The data is now in the disk file.
 For Lotus 1-2-3, use the /File Import Numbers
 command to read the data.
 For a database manager, use the appropriate
 import ASCII data command.
*/
}
```

```
/*
      Example 4: use of TRANSMIT
*/

#include <ieee-c.h>

main()
{
      int status;
      initialize (21,0);
/*
 Now, send a trigger to three devices at addresses
 2,4, and 5, using the LISTEN and GET (group execute
 trigger) commands.
*/
      transmit ("REN UNL LISTEN 2 4 5 GET",&status);
}
```

```
/*
      Example 5: receiving binary array data
            (from a Tek 7612D digitizer)
*/

#include <ieee-c.h>

char sdc[] = "REN LISTEN 1 SEC 0 SDC";
int data[1024];

main ()
{
  int status,l;

  initialize (21,0);
  transmit (sdc,&status);  /* selected device clear */

  /* start digitizing */
  transmit ("MTA DATA 'REP 0,A' END",&status);
  /* set device to talk */
  transmit ("MLA TALK 1 SEC 0",&status);
  /* read 3 bytes of header info */
  rarray (data,3,&l,&status);
  /* read 2048 bytes of waveform data */
  rarray (data,2048,&l,&status);
  /* note: the device sends data in a non-PC format */
  /* clear device to stop it */
  transmit (sdc,&status);
}
```

Oh-the hardware's connected to the firmware,
and the firmware's connected to the software,
and the software's connected to the liveware,
all the live-long day.
- Kerry Newcom

The IEEE Standard 488-1978 is a byte-serial, bit parallel asynchronous interface originally defined for programmable measurement instrument systems. Because it is easy to use and allows flexibility in communicating data and control information, it has become a common interface for computer peripherals as well as instruments. The standard defines the electrical specifications, cables, connectors, control protocol, and commands required to allow data transfer between devices.

The IEEE-488 is also referred to a ANSI MC1.1 and IEC 625.1. All three are identical except for the IEC 625.1 which uses a slightly different connector. The IEEE 488 standard is also commonly referred to by a manufacturers' brand name. These brand names include HP-IB, GP-IB, IEEE BUS, ASCII BUS, and PLUS BUS. All of the brand name implementations are mechanically and electrically identical and will be referred to by the abbreviation gp-ib.

Even though a wide range of instruments can be attached to the bus, system configuration is straightforward because all specifications are precisely defined in terms of their electrical, mechanical, and functional requirements. This allows equipment from different manufacturers to be connected at relatively low cost with few restrictions on data rates and communications protocol. However, the system has the following defined constraints:

- No more than 15 devices can be interconnected by a single bus.

- Total transmission length cannot exceed 20 meters, or 2 meters times the number of devices, whichever is less.

- Data rate through any signal line must be less than or equal to one megabit/second.

Although the interface was originally designed for instruments, it has become a well received standard for computer peripheral communication. A computer, acting as the active controller of the bus, can logically address up to 31 primary devices each with up to 31 secondary addresses. The peripherals may be connected in either a star or linear topology.

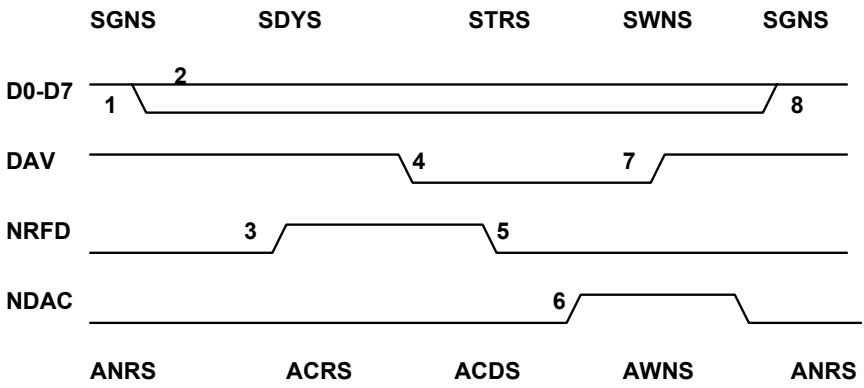
There may be more than one device with controller capability but there can be only one active controller at a time. Any controller can pass control to another controller but only the system controller can unconditionally assume control of the bus.

The maximum IEEE-488.1 data rate is one megabyte per second over limited distances and typically five to twenty kilobytes per second over the full transmission path. The bus uses a three wire handshake to coordinate data and command transfers and every transmitted byte undergoes a handshake. This method guarantees data transfer timing integrity among devices that may be operating at different transfer rates. It also imposes the restriction that data will be transferred at the rate of the slowest device involved in the transaction. This scheme for tranferring data is patented by Hewlett-Packard.

The IEEE-488SD specification uses a modified two-wire handshake to allow data rates up to 5 megabytes per second over limited distances.

The bus transmitter and receiver specifications are generally TTL compatible, however, several manufacturers have designed special parts that improve bus performance over standard TTL devices. These improvements include receiver hysteresis to reduce noise susceptibility, high impedance inputs, bus terminating resistors, and no loading of the bus when the device is powered down. The IEEE-488 drivers (75160 and 75162) incorporate these improvements.

Handshake Timing Sequence



The timing diagram relates the electrical signals on the bus to the states of the source and acceptor handshakes. By looking at both, it may be easier to relate the hardware handshake to IEEE-488 state diagrams.

1. Initially, the source goes to the source generate state (SGNS). In SGNS the source is not asserting the data lines or data valid (DAV). In the passive state the data lines rise to a high level. The acceptors are in the acceptor not ready state (ANRS), with both not ready for data (NRFD) and not data accepted (NDAC) asserted.

2. The source asserts the data lines and enters the source delay state (SDYS). If this is the last data byte in a message, the source may also assert end or identify (EOI). The source waits for the data to settle on the data lines and for all acceptors to reach the acceptor ready state (ACRS).

3. Each acceptor releases its not ready for data (NRFD) line and moves to the acceptor ready state (ACRS). Any acceptor can delay the handshake by not releasing NRFD.

4. When the source sees NRFD high, it enters the source transfer state (STRS) by asserting data valid (DAV).

5. When the receiver(s) see that DAV is asserted, they enter the accept data state (ACDS). Each device then asserts NRFD since it is busy with the current data byte.

6. As the devices accept data they release NDAC to move from the ACDS to the acceptor wait for new cycle state (AWNS). All receivers must release the NDAC line before the source can move to the next state (SWNS).

7. When NDAC is high, the source wait for new cycle state (SWNS) is entered. In SWNS, the source releases DAV. The acceptors then enter their initial state (acceptor not ready state ANRS).

8. The source returns to its initial state (source generate SGNS) and the cycle resumes.

The bus consists of 24 wires, 16 of which are information transmission lines.

- Data bus - eight bidirectional data lines.
- Transfer bus - three data transfer control lines.
- Management bus - five interface management lines.

The eight remaining lines are ground and one of these may be designated as the cable shield ground.

The cable shield ground on your board is connected to digital ground through a jumper near the connector. This jumper may be removed to allow the shield ground to be connected to chassis ground at the rear panel connecting screw. The jumper can be also be removed to allow an instrument to supply the shield ground. In most applications the cable must be shielded to comply with FCC regulations for computing equipment. The cable connectors are designed to be stacked and cables come in various lengths (.5, 1, 2, and 4 meter lengths) to accommodate most system configurations.

Data Bus

DIO1 through DIO8 - bidirectional asynchronous data lines.

Management Bus

ATN (Attention) - a bus management line that indicates whether the current data is to be interpreted as data or a command. When asserted with EOI it indicates that a parallel poll is in process.

EOI (End or identify) - a bus management line that indicates the termination of a data transfer. When asserted with ATN, it indicates that a parallel poll is in process.

IFC (Interface Clear) - a bus management line asserted only by the system controller to take unconditional control of the bus. The bus is cleared to a quiescent state and all talkers and listeners are placed in an idle state.

REN (Remote enable) - a bus management line that allows instruments on the bus to be programmed by the active controller (as opposed to being programmed only through the instrument controls).

SRQ (Service request) - a bus management line used by a device to request service from the active controller.

Transfer Bus

DAV (Data Valid) - one of three handshake lines used to indicate availablility and validity of information on the DIO lines.

NDAC (Not Data Accepted) - a handshake line used to indicate the acceptance of data by all devices.

NRFD (Not Ready For Data) - a handshake line used to indicate that all devices are not ready to accept data.

The **star** cabling topology minimizes worst-case transmission path lengths but concentrates the system capacitance at a single node.

The **linear** cabling topology produces longer path lengths but distributes the capacitive load. Combinations of star and linear cabling configurations are also acceptable.

BASICA and GWBASIC Language Interface

This section documents the use of BASICA or GWBASIC with our IEEE-488 interfaces.

IEEE-488 routines are called in BASIC and BASICA with the CALL statement. Before CALL can be used, the software must be loaded by running the BASIC488.EXE program:

```
C:\CEC488>  BASIC488
```

BASIC488 can be added to your AUTOEXEC.BAT file so it will run automatically when the computer is turned on.

Then, DEF SEG must be used to indicate the memory segment address of the software. A variable named for each IEEE-488 routine must be set to the offset for that routine (example: INITIALIZE=0). Variables used with the calls must be integers (names end with a percent sign), or strings (names end with a dollar sign).

The following code shows how to set up BASIC to be able to call 488 routines:

```
10 DEF SEG=&H0
12 IEEE=PEEK(&H182)+256*PEEK(&H183)
14 DEF SEG=IEEE
16 IF IEEE=0 THEN STOP
20 INITIALIZE=0
```

Subroutine offset variables, as in line 20, only need to be set once in the program.

Note: if you have programs written using the older method of BASIC programming, where the DEF SEG statement refers directly to the on-board ROM of the '488 card, they will still work. (However, note some newer versions of the card have no ROM). There is no need to modify old programs. However, using the new method only requires changing the first few lines of the program and allows access to additional routines such as SETTIMEOUT.

The following pages give the calling information for each routine.

INITIALIZE

```
INITIALIZE = 0
CALL INITIALIZE (my.addr%,level%)
```

- "my.addr%" is the GPIB address to be used by the board. It must be in the range 0 to 30.
- "level%" indicates whether the board should be a system controller. Use 0 for system controller (send interface clear), and 2 for device.

SEND

```
SEND = 9
CALL SEND (addr%,info$,status%)
```

- "addr%" is the GPIB address of the device to send the data to. (must be 0 to 30).
- "info$" is the data string to be transmitted.
- "status%" is returned after the call.

ENTER

```
ENTER = 21
recv$ = SPACE$(80)
CALL ENTER (recv$,length%,address%,status%)
recv$ = LEFT$(recv$,length%)
```

- "recv$" is returned after the call, containing the received data. SPACE$(max.length) assigns a string of spaces to the receive string. recv$ must be set to a string whose length is greater than or equal to the number of characters expected. recv$ should be trimmed to length with the LEFT$ function.
- "length%" is returned after the call and is equal to the length of the received string.
- "address%" is the GPIB address of the designated talker.
- "status%" is returned after the call.

SPOLL

```
SPOLL = 12
CALL SPOLL (address%,poll%,status%)
```

- "address%" is the GPIB address of the device to be polled.
- "poll%" is returned after the call, containing the poll result value.
- "status%" is returned after the call.

PPOLL

```
PPOLL = 15
CALL PPOLL (poll%)
```

- "poll%" is returned after the call, containing the poll result value.

TRANSMIT

```
TRANSMIT = 3
CALL TRANSMIT (command$,status%)
```

- "command$" is a string containing a sequence of GPIB commands, separated by one or more spaces. (for example, "UNT UNL LISTEN 1"). See chapter 3..
- "status%" is returned after the call with one or more bits set to one to indicate various error conditions.

RECEIVE

```
RECEIVE = 6
recv$ = SPACE$(80)
CALL RECEIVE (recv$,length%,status%)
recv$ = LEFT$(recv$,length%)
```

- "recv$" is returned after the call, containing the received data. SPACE$(max.length) assigns a string of spaces to the receive string. recv$ must be set to a string whose length is greater than or equal to the number of characters expected. recv$ should be trimmed to length with the LEFT$ function.
- "length%" is returned after the call and is equal to the length of the received string.
- "status%" is returned after the call.

```
TARRAY = 200
CALL TARRAY (seg%,ofs%,count%,eoi%,status%)
```

- "seg%" is the segment portion of the memory address of the data. seg% is usually set to -1 to indicate the default data segment.
- "ofs%" is the offset portion of the memory address of the data. This is usually obtained with the VARPTR function.
- "count%" is the number of bytes to be transmitted.
  Note: In BASIC, when you want to send more than 32767 bytes, you will have to assign the value to count% in hex. Example: COUNT%=&HA000
- "eoi%" indicates whether or not to send EOI with the last data byte. 0=NO, 1=YES.
- "status%" indicates whether the transfer went OK.

Memory addresses in BASIC

In BASIC, both **Tarray** and **Rarray** require the memory address of the data area as the first two arguments. This address is given in two parts, due to the nature of addressing on the PC's microprocessor.

SEG% indicates the segment portion of the memory address. This is the upper 16 bits out of the total 20 bits that form a PC memory address. SEG% can also take on the special value -1 to indicate that the default data segment should be used. The default data segment in BASIC contains all variables and arrays in the program. Care should be taken if an absolute segment address is used, since it is possible to access any portion of the computer's memory in this way.

OFS% indicates the offset portion of the memory address. This is the lower 16 bits out of the total 20 bits. Note that this does overlap the segment portion of the address. Both are added together to make up the entire address. If a BASIC array variable is being used, the VARPTR function will return the offset portion of the address.

WARNING: The BASIC interpreter may move some variables to new locations in memory any time a brand new variable is introduced in the program. For example, if the array INFO% shown above was at an offset of &H1000, and the statement "I=4" was executed, where "I" is a new variable, INFO% might be moved to &H1010. If BASIC does this, it can invalidate the OFS% value you obtained with the VARPTR function.

Calling **Tarray** with an invalid offset value could write over memory and cause an error. To avoid this, make sure that VARPTR is the last step before the **Tarray** or **Rarray** call, and that ALL variables used inside the call have previously occurred in the program.

RARRAY

```
RARRAY = 203
CALL RARRAY (seg%,ofs%,count%,length%,status%)
```

- "seg%" is the segment portion of the memory address of the data. seg% is usually set to -1 to indicate the default data segment.
- "ofs%" is the offset portion of the memory address of the data. This is usually obtained with the VARPTR function.
- "count%" is the max. number of bytes to be received.
- "length%" is returned after the call, containing the actual number of data bytes received.
- "status%" indicates whether the transfer went OK.

SRQTEST

```
SRQTEST = 63
CALL SRQTEST(s%)
```

- "s%" indicates whether an SRQ signal has been received.

SETPORT

```
SETPORT = 57
CALL SETPORT(board%,port%)
```

- "board%" is a number from 0 to 3, indicating which '488 board is being set.
- "port%" is the I/O base address of the board. The default settings are &H2B8 for board 0, and boards 1-3 not available.

BOARDSELECT

```
BOARDSELECT = 60
CALL BOARDSELECT(board%)
```

- "board%" is a number from 0 to 3, indicating which '488 board is being selected. This board will be accessed for subsequent CALLs.

DMACHANNEL

```
DMACHANNEL = 45
CALL DMACHANNEL (ch%)
```

- "ch%" is the DMA channel number, and must match the hardware jumper setting of the board. (On PS<>488, channel number 1 can always be used, because the software handles channel assignments automatically). A channel number of -1 disables DMA. The default is DMA disabled.

SETTIMEOUT

```
SETTIMEOUT = 48
CALL SETTIMEOUT (t%)
```

- "t%" is the new timeout value in milliseconds, from 55 to 65535 (&HFFFF).

SETOUTPUTEOS

```
SETOUTPUTEOS = 54
CALL SETOUTPUTEOS (e1%,e2%)
```

- "e1%" is the ASCII code value for the first end-of-string character to be used by the SEND routine.
- "e2%" is the code for the second EOS character. If e2% is 0, only one EOS character is used. The default is e1%=10, e2%=0.

SETINPUTEOS

```
SETINPUTEOS = 51
CALL SETINPUTEOS (e%)
```

- "e%" is the end-of-string character which will terminate reception in the RECEIVE and ENTER routines. The default is 10 (line feed).

LISTENER.PRESENT

```
LISTENER.PRESENT = 84
CALL LISTENER.PRESENT (addr%,present%)
```

- "addr%" is the address to be tested.
- "present%" returns 1 if a listener is present at that address.

BOARD.PRESENT

```
BOARD.PRESENT = 87
CALL BOARD.PRESENT (present%)
```

- "present%" returns 0 if no board is found, non-zero if the board is present.

ENABLE.488EX

```
ENABLE.488EX = 90
CALL ENABLE.488EX (e%)
```

- "e%" should be 1 to enable 488EX enhancements (default=1), or 0 to disable them and treat 488EX as if it was PC488. This could be useful if a device has a problem which doesn't allow it to handle the full speed IEEE-488 data rate correctly.

GPIBFeature

```
GPIBFeature = 96
CALL GPIBFeature (f%,result%)
```

- "f%" is the feature number you wish to test (see the description of the GPIBFeature routine in the programming chapter).
- "result%" is the returned value

Examples

Using INITIALIZE, SEND, and ENTER for a simple measurement:

```
10 DEF SEG=&H0
12 IEEE=PEEK(&H182)+256*PEEK(&H183)
14 DEF SEG=IEEE
16 IF IEEE=0 THEN STOP
20 INITIALIZE=0 : SEND=9 : ENTER=21
30 '
40 my.addr%=21 : level%=0
50 CALL INITIALIZE (my.addr$,level%)
60 '
70 s$="F0R0X" : addr%=16
80 CALL SEND (addr%,s$,status%)
90 r$=SPACE$(80)
100 CALL ENTER (r$,l%,addr%,status%)
110 r$=LEFT$(r$,l%)
120 PRINT "Received data is '";r$;"'"
```

Using TRANSMIT and RARRAY to receive binary data:

```
5 DIM D%(1000)
10 DEF SEG=&H0
12 IEEE=PEEK(&H182)+256*PEEK(&H183)
14 DEF SEG=IEEE
16 IF IEEE=0 THEN STOP
30 INITIALIZE=0 : TRANSMIT=3 : RARRAY=203
40 ' initialize the GPIB
50 my.addr%=21 : level%=0
60 CALL INITIALIZE (my.addr%,level%)
70 ' send a command to the device
80 cmd$="REN MTA LISTEN 3 DATA 'READ' END"
90 CALL TRANSMIT (cmd$,status%)
100 ' set device to talk and read the data
110 cmd$="MLA TALK 3"
120 CALL TRANSMIT (cmd$,status%)
130 count%=2000 : l%=0 : s%=-1
140 o%=VARPTR(d%(1))
150 CALL RARRAY (s%,o%,count%,l%,status%)
160 END
```

QuickBASIC, QBASIC, and Visual BASIC Language Interface

Interface files and setup

Visual BASIC (Windows, Win95, WinNT)

You will need the following files from the application diskette (installed in the CEC488 directory as part of setup):

VB\IEEEVB.BAS	**Add this file to your project using the File / Add File menu command**
WINDOWS\WIN488.DLL	if you are using Windows 3.x (must be in the Windows directory)
WIN32\IEEE_32M.DLL	if you are using Win95 or WinNT (must be in the Windows directory)

Note: if you are using VB 3.0, use the file IEEEVB3.BAS instead.

QBASIC

You will need the following files:

BASIC\BASIC488.EXE	Run this program before running QBASIC.
QBASIC\IEEEQBAS.BAS	Load this file using File/Open and then Save As a new file name.

QuickBASIC 4.x

You will need the following files, which you should copy to the working directory where you will be developing your program:

QB\IEEEQB.QLB	Load this file when you start QB, by using the following DOS command line: C:> QB /L IEEEQB.QLB
QB\IEEEQB.LIB	This file is used when you create EXE files.
QB\IEEEQB.BI	Add the following line to your program: '$INCLUDE: 'IEEEQB.BI'

Earlier versions of QuickBASIC can be used: see the file QB\QB2OR3.DOC on the disk. However, support is limited and not all routines can be called.

BASIC 7.0

Read the BASIC7\BASIC7.DOC file provided on the disk.

Other BASIC versions

Borland Turbo BASIC (a long obsolete language version) is supported using direct ROM access (this will NOT work on cards like the PCI card with no ROM). See the file TURBOBAS\TB488.DOC.

The following code shows the calling sequence for each IEEE-488 interface subroutine. Those arguments which are labelled as (VALUE) may be passed as constants rather than variables if you wish. For example, you can call INITIALIZE as either:

```
CALL INITIALIZE (my.addr%,level%)
```

or

```
CALL INITIALIZE (21,0)
```

Note that integer variable names end with a percent sign (%) and that integer constants do not contain a decimal point.

INITIALIZE

```
CALL INITIALIZE (my.addr%,level%)
```

- "my.addr%" (VALUE) is the IEEE-488 address to be used by the interface card. It must be an integer from 0 to 30, and should differ from the addresses of all devices to be connected.
- "level%" (VALUE) indicates whether or not the interface card will be the system controller. It should be zero for system control mode and two for device mode.

SEND

```
CALL SEND (addr%,info$,status%)
```

- "addr%" (VALUE) is an integer indicating the IEEE-488 address of the device to send the data to.
- "info$" (VALUE) is the data to be sent. One or two end-of-string characters may be added to the data in info$ (see SETOUTPUTEOS later, the default is line feed).
- "status%" indicates the success or failure of the data transfer.

ENTER

For QuickBASIC and QBASIC:

```
r$ = SPACE$(80)
CALL ENTER (r$,l%,addr%,status%)
r$ = LEFT$(r$,l%)
```

For Visual BASIC:

```
CALL ENTER (r$,maxlen%,l%,addr%,status%)
```

- "r$" is the string into which the received data will be placed. In DOS QuickBASIC or QBASIC, space for the received data must be allocated, usually with the SPACE$ function, as shown. The desired maximum number of received characters is used as an argument to the SPACE$ function. r$ should be trimmed to the actual received length with the LEFT$ function after the ENTER call.
- "maxlen%" (VALUE) is the maximum number of characters to receive. This argument is not present in the DOS QuickBASIC or QBASIC language interface.
- "l%" is a variable which will be set to the actual received length.
- "addr%" (VALUE) is the IEEE-488 address of the device to read from.
- "status%" indicates the success or failure of the data transfer.

SPOLL

```
CALL SPOLL (addr%,poll%,status%)
```

- "addr%" (VALUE) is an integer indicating the IEEE-488 address of the device to serial poll.
- "poll%" is a variable which will be set to the poll result.
- "status%" indicates the success or failure of the operation.

PPOLL

```
CALL PPOLL (poll%)
```

- "poll%" is a variable which will be set to the result of the parallel poll operation.

TRANSMIT

```
CALL TRANSMIT (cmd$,status%)
```

- "cmd$" (VALUE) is a string containing a sequence of IEEE-488 commands and data. See chapter 3.
- "status%" indicates the success or failure of the operation.

RECEIVE

For QuickBASIC and QBASIC:

```
r$ = SPACE$(80)
CALL RECEIVE (r$,l%,status%)
r$ = LEFT$(r$,l%)
```

For Visual BASIC:

```
CALL RECEIVE (r$,maxlen%,l%,status%)
```

- "r$" is the string into which the received data will be placed. In DOS QuickBASIC or QBASIC, space for the received data must be allocated, usually with the SPACE$ function, as shown. The desired maximum number of received characters is used as an argument to the SPACE$ function. r$ should be trimmed to the actual received length with the LEFT$ function after the RECEIVE call.
- "maxlen%" (VALUE) is the maximum number of characters to receive. This argument is not present in the DOS QuickBASIC or QBASIC language interface.
- "l%" is a variable which will be set to the actual received length.
- "status%" indicates the success or failure of the data transfer.

TARRAY

Note that this routine is **not** available in DOS QBASIC.

```
CALL TARRAY (d%(1),count%,eoi%,status%)
```

- "d%" is the array variable containing the data to be transmitted.
- "count%" (VALUE) is the number of bytes to be transmitted.
- "eoi%" (VALUE) is zero if the EOI signal is not desired on the last byte, or one if it is desired.

- "status%" indicates the success or failure of the operation.

RARRAY

Note that this routine is **not** available in DOS QBASIC.

```
CALL RARRAY (d%(1),count%,l%,status%)
```

- "d%" is the array variable into which data will be received.
- "count%" (VALUE) is the maximum number of bytes to be received.
- "l%" is a variable which will be set to the actual number of bytes received.
- "status%" indicates the success or failure of the operation.

SRQ%

```
IF (SRQ%) THEN ... ' put any statement here
```

or

```
WHILE (NOT (SRQ%)) ' wait for SRQ
WEND
```

SETPORT

Note: this routine is not needed on plug-and-play boards like the PCI and Microchannel bus interface cards, since the I/O port address is handled automatically. It needed on the other 488 interfaces only if the I/O address is changed from the standard setting of 2B8 hex.
Note also that you can leave this call out and simply change the configuration file CEC488.INI (see entry on configuration files in the programming chapter).

```
CALL SETPORT (board%,ioport%)
```

- "board%" (VALUE) is the IEEE-488 board number (from 0 to 3). Use zero if you have only one IEEE-488 board.
- "ioport%" (VALUE) is the I/O port address of the IEEE-488 board.

BOARDSELECT

```
CALL BOARDSELECT (board%)
```

- "board%" (VALUE) is the IEEE-488 board number (from 0 to 3).

DMACHANNEL

```
CALL DMACHANNEL (ch%)
```

- "ch%" (VALUE) is the DMA (direct memory access) channel to be used by the 488 interface board. If -1 is used, DMA is disabled for the current board. The default setting is -1 (disabled).

SETTIMEOUT

```
CALL SETTIMEOUT (msec%)
```

- "msec%" (VALUE) is the desired timeout period in milliseconds.

SETOUTPUTEOS

```
CALL SETOUTPUTEOS (eos1%,eos2%)
```

- "eos1%" and "eos2%" (VALUE) are the desired end-of-string characters to be appended when the SEND call is used. If only one end-of-string character is desired, set eos2% to zero.

SETINPUTEOS

```
CALL SETINPUTEOS (eos%)
```

- "eos%" (VALUE) is the desired end-of-string character which will cause the ENTER and RECEIVE routines to terminate.

LISTENERPRESENT%

```
present% = ListenerPresent%(addr%)
```

- "addr%" is the GPIB address to be tested.
- "present%" is the return value, which indicates whether a listener was present at that address.

GPIB BOARD PRESENT%

```
IF NOT(GpibBoardPresent%) THEN ...
```

ENABLE 488EX

```
CALL Enable488EX (e%)
```

- "e%" is 1 to enable 488EX enhancements, 0 to disable them. The default is enabled. This call has no effect on other 488 boards.

Using Tarray and Rarray with strings in DOS QuickBASIC

In QuickBASIC, the **Tarray** and **Rarray** routines are designed to take numeric array variables as arguments. Sometimes, however, it is useful to be able to pass string variables to them. In QuickBASIC, strings are stored very differently from arrays, so you cannot just put a string variable in place of the array. You can use the DECLARE statements shown in the example below to set up new routine names, **Tarraystr** and **Rarraystr**, which refer to the same '488 routines, but take different arguments. Then, pass the string variable by using the QuickBASIC VARSEG, SADD, and LEN functions, which get the variable's address and length.

```
' Declare TARRAYSTR and RARRAYSTR for string variables

DECLARE SUB tarraystr ALIAS "tarray" (
      byval s as integer, byval o as integer,
      count as integer, eoi as integer,
      status as integer)
DECLARE SUB rarraystr ALIAS "rarray" (
      byval s as integer, byval o as integer,
      count as integer, 1 as integer,
      status as integer)

' Example of using Tarraystr:

CALL TRANSMIT ("MTA LISTEN 8",status%) ' set up
transfer
S$ = "sample data string"
CALL TARRAYSTR (VARSEG(S$),SADD(S$),LEN(S$),1,status%)
```

Examples - QuickBASIC or QBASIC

Using INITIALIZE, SEND, and ENTER for a simple measurement:

```
' $INCLUDE: 'ieeeqb.bi' '(for QB 4.x only)
CALL INITIALIZE (21,0)
CALL SEND (16,"F0R0X",status%)
r$ = SPACE$(80)
CALL ENTER (r$,l%,16,status%)
r$ = LEFT$(r$,l%)
PRINT "Received data is '";r$;"'"
```

Using TRANSMIT and RARRAY to receive binary data: (code shown for QuickBASIC 4.x)

```
' $INCLUDE : 'ieeeqb.bi'
DIM D%(1000)
'
' initialize the GPIB
'
CALL INITIALIZE (21,0)
'
' send a command to the device
'
CALL TRANSMIT ("REN MTA LISTEN 3 DATA 'READ'
                  END",status%)
'
' set device to talk and read the data
'
CALL TRANSMIT ("MLA TALK 3",status%)
CALL RARRAY (d%(1),2000,l%,status%)
```

Examples - Visual BASIC

Using INITIALIZE, SEND, and ENTER for a simple measurement:

```
CALL INITIALIZE (21,0)
CALL SEND (16,"F0R0X",status%)
CALL ENTER (r$,80,l%,16,status%)
PRINT "Received data is '";r$;"'"
```

Using TRANSMIT and RARRAY to receive binary data:

```
DIM D%(1000)
'
' initialize the GPIB
'
CALL INITIALIZE (21,0)
'
' send a command to the device
'
CALL TRANSMIT ("REN MTA LISTEN 3 DATA 'READ'
                END",status%)
'
' set device to talk and read the data
'
CALL TRANSMIT ("MLA TALK 3",status%)
CALL RARRAY (d%(1),2000,l%,status%)
```

Much older (pre-1989) manuals showed the use of the CALL ABSOLUTE statement in QuickBASIC. This has been replaced by the CALL statement.

However, existing programs written with CALL ABSOLUTE will still work on new boards. There is no requirement to modify old programs. Note that these old programs will contain a DEF SEG statement which refers to the memory address of the ROM on the 488 board. It is important that the DEF SEG statement match the hardware setting of the board.

Note that newer versions of the boards have no ROM. The older ROM-based boards are now obsolete, and it is strongly recommended that you update your software. This is quite easy, and technical assistance is available. A very limited number of older ROM-based boards will be available for some period of time - contact the factory for details.

Programs written with the current software will also run on older boards. We recommend that you use the method in this manual for new programs.

Turbo Pascal and Delphi Language Interface

Interface files

Turbo Pascal (DOS)

The Turbo Pascal support files are located in directory \turbopas on the IEEE-488 applications disk. You should copy the necessary files to a working directory on your system disk. You will need:

TURBOPAS\IEEEPAS.PAS
TURBOPAS\PAS488.OBJ

Next, you need to compile the interface code to create a Turbo Pascal UNIT file (TPU). Load IEEEPAS.PAS into Turbo Pascal. Select the Compile menu, set the destination to "Disk", and select "Build".

Note: only Turbo or Borland Pascal v7 is currently supported. If you have an earlier version of Turbo Pascal, you must obtain an earlier application disk (v4.07). All the routines except the GPIBFeature() routine are supported with this older disk.

Now put the following line in your program:

```
uses ieeepas;
```

Borland Delphi (32-bit) for Windows

You will need the following files, provided on the application disk and installed as part of normal SETUP:

DELPHI\IEEEDEL.PAS	**Add this file to your project.**
WIN32\IEEE32_M.DLL	This file must be in your Windows directory (it is installed there by SETUP)

Now put the following line in your program:

```
uses ieeedel;
```

The following code shows the calling sequence for each IEEE-488 interface subroutine. Those arguments which are labelled as (VALUE) may be passed as constants rather than variables if you wish. For example, you can call INITIALIZE as either:

```
initialize (my_addr,level);
```

or

```
initialize (21,0);
```

Note that integer constants do not contain a decimal point.

INITIALIZE

```
initialize (my_addr,level);
```

- "my_addr" (integer VALUE) is the IEEE-488 address to be used by the interface card. It must be an integer from 0 to 30, and should differ from the addresses of all devices to be connected.
- "level" (integer VALUE) indicates whether or not the interface card will be the system controller. It should be zero for system control mode and two for device mode.

SEND

```
send (addr,info,status);
```

- "addr" (integer VALUE) is an integer indicating the IEEE-488 address of the device to send the data to.
- "info" (string VALUE) is a string containing the data to be sent. One or two end-of-string characters may be added to the data (see SETOUTPUTEOS later, the default is line feed).
- "status" (integer) is a variable which indicates the success or failure of the data transfer.

ENTER

```
enter (rstring,maxlen,1,addr,status);
```

- "rstring" (string) is the string into which the received data will be placed.
- "maxlen" (word VALUE) is the maximum number of characters desired.
- "l" (word) is a variable which will be set to the actual received length.
- "addr" (integer VALUE) is the IEEE-488 address of the device to read from.
- "status" (integer) is a variable which indicates the success or failure of the data transfer.

SPOLL

```
spoll (addr,poll,status);
```

- "addr" (integer VALUE) is an integer indicating the IEEE-488 address of the device to serial poll.
- "poll" (byte) is a variable which will be set to the poll result.
- "status" (integer) is a variable which indicates the success or failure of the data transfer.

PPOLL

```
ppoll (poll);
```

- "poll" (byte) is a variable which will be set to the result of the parallel poll operation.

TRANSMIT

```
transmit (cmdstring,status);
```

- "cmdstring" (string VALUE) is a string containing a sequence of IEEE-488 commands and data. See chapter 3.
- "status" (integer) is a variable which indicates the success or failure of the data transfer.

RECEIVE

```
receive (rstring,maxlen,l,status);
```

- "rstring" (string) is the string into which the received data will be placed.
- "maxlen" (word VALUE) is the maximum number of characters desired.
- "l" (word) is a variable which will be set to the actual received length.
- "status" (integer) is a variable which indicates the success or failure of the data transfer.

TARRAY

```
tarray (d,count,eoi,status);
```

- "d" (any type) is the variable containing the data to be transmitted.
- "count" (word VALUE) is the number of bytes to be transmitted.
- "eoi" (boolean VALUE) is FALSE if the EOI signal is not desired on the last byte, or TRUE if it is desired.
- "status" (integer) is a variable which indicates the success or failure of the data transfer.

RARRAY

```
rarray (d,count,l,status);
```

- "d" (any type) is the variable into which data will be received. "count" (word VALUE) is the maximum number of bytes to be received.
- "l" (word) is a variable which will be set to the actual number of bytes received.
- "status" (integer) is a variable which indicates the success or failure of the data transfer.

SRQ

```
IF (srq) THEN ... { put any statement here }
```

SETPORT

Note: this routine is not needed on plug-and-play boards like the PCI and Microchannel bus interface cards, since the I/O port address is handled automatically. It needed on the other 488 interfaces only if the I/O address is changed from the standard setting of 2B8 hex.
Note also that you can leave this call out and simply change the

configuration file CEC488.INI (see entry on configuration files in the programming chapter).

```
setport (board,ioport);
```

- "board" (integer VALUE) is the IEEE-488 board number (from 0 to 3). Use zero if you have only one IEEE-488 board.
- "ioport" (word VALUE) is the I/O port address of the IEEE-488 board.

BOARDSELECT

```
boardselect (board);
```

- "board" (integer VALUE) is the IEEE-488 board number (from 0 to 3).

DMACHANNEL

```
dmachannel (ch);
```

- "ch" (integer VALUE) is the DMA (direct memory access) channel to be used by the 488 interface board. If -1 is used, DMA is disabled for the current board. The default setting is -1 (disabled).

SETTIMEOUT

```
settimeout (msec);
```

- "msec" (word VALUE) is the desired timeout period in milliseconds.

SETOUTPUTEOS

```
setoutputEOS (eos1,eos2);
```

- "eos1" and "eos2" (byte VALUE) are the desired end-of-string characters to be appended when the SEND call is used. If only one end-of-string character is desired, set eos2 to chr(0).

SETINPUTEOS

```
setinputEOS (eos);
```

- "eos" (byte VALUE) is the desired end-of-string character which will cause the ENTER and RECEIVE routines to terminate.

LISTENER PRESENT

```
present = listener_present (addr);
```

- "addr" (integer VALUE) is the address to test
- the return value (boolean) indicates whether a listener is present.

GPIB BOARD PRESENT

```
if (gpib_board_present = 0) then ...
```

ENABLE 488EX

```
enable_488ex (e);
```

"e" (boolean VALUE) enables or disables 488EX enhancements.

GPIBFeature

```
value = GPIBFeature (feature);
```

- "feature" (integer VALUE) is the feature you wish to inquire about. You can use various predefined constants in the IEEEPAS unit.
- the return value (integer) is the requested information

Examples

Using INITIALIZE, SEND, and ENTER for a simple measurement:

```
PROGRAM example;
USES ieeepas;
VAR
      status : integer;
      l : word;
      r : string;
BEGIN
      initialize (21,0);
      send (16,'F0R0X',status);
      enter (r,l,16,status);
      writeln ('Received data is ''',r,'''');
END.
```

Using TRANSMIT and RARRAY to receive binary data:

```
PROGRAM example2;
USES ieeepas;
VAR
      status : integer;
      l : word;
      d : array[1..1000] of integer;
BEGIN
      initialize (21,0);
      { send a command to the device }
transmit ('REN MTA LISTEN 3 DATA ''READ''
END',status);
 { set device to talk and read the data }
      transmit ('MLA TALK 3',status);
      rarray (d,2000,l,status);
END.
```

C and C++ Language Interface

This manual describes the use of C with our IEEE-488 interfaces. This software has been tested with Microsoft C, both DOS and Windows (Visual C++), and with Borland's Turbo C and C++. The same software also works with other C compilers, as long as they support the keywords "pascal" and "far", and Microsoft library format.

Interface files

The C support files are located in directory \C on the IEEE-488 applications disk, and are installed as part of the normal SETUP. You should copy the necessary files to the working directory you will be using to develop your program.

You will need:

C\IEEE-C.H	**Include this file in your source code: #include "ieee-c.h"**
IEEE488.LIB	Link this library for **DOS** programs.
WIN488.LIB	Link this for **16-bit Windows** programs.
IEEE_32M.LIB	Link this when using Microsoft Visual C++ for **Win32** programs (95 or NT).
IEEE_32B.LIB	Link this when using Borland C++ for **Win32** programs (95 or NT).

(to create 16-bit programs for the OS/2 operating system, use IEEEOS2.LIB)

First, write your C program using the IEEE-488 subroutine calls shown in the next section. Make sure to include the line:

```
#include "ieee-c.h"
```

Compile your program normally.
You must link with the appropriate library as listed earlier.

Microsoft C, for DOS

Use a command line like this to compile and link your program:

```
CL  myprog.c   /link  ieee488.lib
```

Borland C/C++ for DOS

Use a command line like this to compile and link your program:

```
BCC  myprog.c  ieee488.lib
```

Microsoft Visual C++, or Borland C++ for Windows

Add the appropriate library file to your project, using the menu commands in the C++ development environment.

for 16-bit Windows:	WIN488.LIB
for 32-bit Windows:	IEEE_32M.LIB or IEEE_32B.LIB (Borland)

(Note that Windows programs require the appropriate DLL file be present when the program is executed. This file (WIN488.DLL for 16-bit, and IEEE_32M.DLL for 32-bit) should be in the Windows directory, where it is installed automatically by the SETUP program).

The following code shows the calling sequence for each IEEE-488 interface subroutine. Those arguments which are not pointers to int's or unsigned's may be passed as constants rather than variables if you wish. For example, you can call SEND as either:

```
send (addr,str,&status);
```

or

```
send (16,"this is a test",&status);
```

Note that integer constants do not contain a decimal point.

INITIALIZE

```
initialize (my_addr,level);
```

- "my_addr" (int) is the IEEE-488 address to be used by the interface card. It must be an integer from 0 to 30, and should differ from the addresses of all devices to be connected.
- "level" (int) indicates whether or not the interface card will be the system controller. It should be zero for system control mode and two for device mode.

SEND

```
send (addr,info,&status);
```

- "addr" (int) is an integer indicating the IEEE-488 address of the device to send the data to.
- "info" (char *) is a string containing the data to be sent. One or two end-of-string characters may be added to the data (see SETOUTPUTEOS later, the default is line feed).
- "status" (int *) is a variable which indicates the success or failure of the data transfer.

ENTER

```
enter (rstring,maxlen,&l,addr,&status);
```

- "rstring" (char *) is the string into which the received data will be placed. The string will automatically have a terminating null byte appended to make a valid C string.
- "maxlen" (unsigned) is the maximum number of characters desired.
- "l" (unsigned *) is a variable which will be set to the actual received length.
- "addr" (int) is the IEEE-488 address of the device to read from.
- "status" (int *) is a variable which indicates the success or failure of the data transfer.

SPOLL

```
spoll (addr,&poll,&status);
```

- "addr" (int) is an integer indicating the IEEE-488 address of the device to serial poll.
- "poll" (char *) is a variable which will be set to the poll result.
- "status" (int *) is a variable which indicates the success or failure of the data transfer.

PPOLL

```
ppoll (&poll);
```

- "poll" (char *) is a variable which will be set to the result of the parallel poll operation.

TRANSMIT

```
transmit (cmdstring,&status);
```

- "cmdstring" (char *) is a string containing a sequence of IEEE-488 commands and data. See chapter 3.
- "status" (int *) is a variable which indicates the success or failure of the data transfer.

RECEIVE

```
receive (rstring,maxlen,&l,&status);
```

- "rstring" (char *) is the string into which the received data will be placed. The string will automatically have a terminating null byte appended to make a valid C string.
- "maxlen" (unsigned) is the maximum number of characters desired.
- "l" (unsigned *) is a variable which will be set to the actual received length.
- "status" (int *) is a variable which indicates the success or failure of the data transfer.

TARRAY

```
tarray (d,count,eoi,&status);
```

- "d" (void *, a pointer to any type) is the array variable containing the data to be transmitted.
- "count" (unsigned) is the number of bytes to be transmitted.
- "eoi" (char) is zero if the EOI signal is not desired on the last byte, or one if it is desired.
- "status" (int *) is a variable which indicates the success or failure of the data transfer.

RARRAY

```
rarray (d,count,&l,&status);
```

"d" (void *, a pointer to any type) is the array variable into which data will be received.

- "count" (unsigned) is the maximum number of bytes to be received.
- "l" (unsigned *) is a variable which will be set to the actual number of bytes received.
- "status" (int *) is a variable which indicates the success or failure of the data transfer.

SRQ

```
if (srq()) ... { put any statement here }
```

SETPORT

Note: this routine is not needed on plug-and-play boards like the PCI and Microchannel bus interface cards, since the I/O port address is handled automatically. It needed on the other 488 interfaces only if the I/O address

is changed from the standard setting of 2B8 hex.
Note also that you can leave this call out and simply change the configuration file CEC488.INI (see entry on configuration files in the programming chapter).

```
setport (board,ioport);
```

- "board" (int) is the IEEE-488 board number (from 0 to 3). Use zero if you have only one IEEE-488 board.
- "ioport" (unsigned) is the I/O port address of the IEEE-488 board.

BOARDSELECT

```
boardselect (board);
```

- "board" (int) is the IEEE-488 board number (from 0 to 3).

DMACHANNEL

```
dmachannel (ch);
```

- "ch" (int) is the DMA (direct memory access) channel to be used by the 488 interface board. If -1 is used, DMA is disabled for the current board. The default setting is -1 (disabled).

SETTIMEOUT

```
settimeout (msec);
```

- "msec" (unsigned) is the desired timeout period in milliseconds.

SETOUTPUTEOS

```
setoutputEOS (eos1,eos2);
```

- "eos1" and "eos2" (char) are the desired end-of-string characters to be appended when the SEND call is used. If only one end-of-string character is desired, set eos2 to zero.

SETINPUTEOS

```
setinputEOS (eos);
```

- "eos" (char) is the desired end-of-string character which will cause the ENTER and RECEIVE routines to terminate.

LISTENER PRESENT

```
present = listener_present (addr);
```

- "addr" (int) is the device address to be tested.
- the return value (int) indicates whether a listener was present.

GPIB BOARD PRESENT

```
if (!gpib_board_present()) ...
```

ENABLE 488EX

```
enable_488ex (e);
```

- "e" (char) is 0 to disable 488EX enhancements, 1 to enable them.

GPIBFeature

```
value = GPIBFeature (feature);
```

- "feature" (int) is the feature you wish to inquire about. You may use any of the predefined constants in the IEEE-C.H file.
- the return value (int) is the information you requested.

Examples

Using INITIALIZE, SEND, and ENTER for a simple measurement:

```
main ()
{
      int status;
      unsigned l;
      char r[80];

      initialize (21,0);
      send (16,"F0R0X",&status);
      enter (r,79,&l,16,&status);
      printf ("Received data is '%s'\n",r);
}
```

Using TRANSMIT and RARRAY to receive binary data:

```
main ()
{
      int status;
      unsigned l;
      int d[1000];

      initialize (21,0);
      /* send a command to the device */
transmit ("REN MTA LISTEN 3 DATA 'READ' END",&status);
      /* set device to talk and read the data */
      transmit ("MLA TALK 3",&status);
      rarray (d,2000,&l,&status);
}
```

Interface files

The FORTRAN support files are located in directory FORTRAN on the IEEE-488 applications disk. You should copy the necessary files to a working directory on your system disk.

Microsoft FORTRAN

Microsoft FORTRAN version 3.3 and later are supported with these files:

FORTRAN\IEEEFOR.OBJ
IEEE488.LIB

For version 5.0 and later, use IEEEFOR5.OBJ instead of IEEEFOR.OBJ.

For Microsoft FORTRAN PowerStation (32-bit), you will need to use the language-independent device driver method in section O of this manual (the CECHP driver). This FORTRAN version is not supported by direct subroutine calls.

Lahey FORTRAN

Use the files:

FORTRAN\IEEEFOR5.OBJ
IEEE488.LIB

RM/FORTRAN

Ryan/McFarland FORTRAN (also marketed as IBM Professional FORTRAN) uses these support files:

FORTRAN\IEEERM.OBJ
IEEE488.LIB

Compiling programs

First, write your FORTRAN program using the IEEE-488 subroutine calls shown in the next section.

Compile your program normally and link it with the supplied object module and library.

Microsoft FORTRAN

```
C> FL myprog.for ieeefor /link ieee488
```

RM/FORTRAN

```
C> RMFORT myprog.for
C> LINK myprog ieeefor,,,ieee488;
```

The following code shows the calling sequence for each IEEE-488 interface subroutine. Those arguments which have the comment (VALUE) may be passed as constants rather than variables if you wish. Note also that integer constants do not contain a decimal point.

Note: String variables are handled differently in each version of FORTRAN.

Microsoft FORTRAN

Use C-style strings, by putting the letter C just after the closing quote of the string constant:

```
CALL SEND (16,'this is a test'C,status)
```

Lahey FORTRAN

For Lahey FORTRAN, you **must** declare all your subroutines with the special keyword **MSEXTERNAL**.
Also, concatenate a null or CHAR(0) byte to each string variable you pass:

```
MSEXTERNAL SEND
CALL SEND (16,'this is a test' // CHAR(0),status)
```

RM/FORTRAN

Just pass the string normally:

```
CALL SEND (16,'this is a test',status)
```

INITIALIZE

```
CALL INITIALIZE (myaddr,level)
```

- INTEGER*2 myaddr
  (VALUE) is the IEEE-488 address to be used by the interface card. It must be an integer from 0 to 30, and should differ from the addresses of all devices to be connected.
- INTEGER*2 level
  (VALUE) indicates whether or not the interface card will be the system controller. It should be zero for system control mode and two for device mode.

SEND

```
CALL SEND (addr,info,status)
```

- INTEGER*2 addr
  (VALUE) is an integer indicating the IEEE-488 address of the device to send the data to.
- CHARACTER*n info
  (VALUE) is a string containing the data to be sent. One or two end-of-string characters may be added to the data (see SETOUTPUTEOS later, the default is line feed). info must be a C-style string (terminated with a null character, see earlier note).
- INTEGER*2 status
  is a variable which indicates the success or failure of the data transfer.

ENTER

```
CALL ENTER (rstring,maxlen,l,addr,status)
```

- CHARACTER*n rstring
  is the string into which the received data will be placed.
- INTEGER*2 maxlen
  (VALUE) is the maximum number of characters desired.
- INTEGER*2 l
  is a variable which will be set to the actual received length.
- INTEGER*2 addr
  (VALUE) is the IEEE-488 address of the device to read from.
- INTEGER*2 status
  is a variable which indicates the success or failure of the data transfer.

SPOLL

```
CALL SPOLL (addr,poll,status)
```

- INTEGER*2 addr
  (VALUE) is an integer indicating the IEEE-488 address of the device to serial poll.
- INTEGER*2 poll
  is a variable which will be set to the poll result.
- INTEGER*2 status
  is a variable which indicates the success or failure of the data transfer.

PPOLL

```
CALL PPOLL (poll)
```

- INTEGER*2 poll
  is a variable which will be set to the result of the parallel poll operation.

TRANSMIT

```
CALL TRANSMIT (cmdstring,status)
```

- CHARACTER*n cmdstring
  (VALUE) is a string containing a sequence of IEEE-488 commands and data. cmdstring must be a C-style string (terminated with a null character, see earlier note).
- INTEGER*2 status
  is a variable which indicates the success or failure of the data transfer.

RECEIVE

```
CALL RECEIVE (rstring,maxlen,l,status)
```

- CHARACTER*n rstring
  is the string into which the received data will be placed.
- INTEGER*2 maxlen
  (VALUE) is the maximum number of characters desired.
- INTEGER*2 l
  is a variable which will be set to the actual received length.
- INTEGER*2 status
  is a variable which indicates the success or failure of the data transfer.

TARRAY

```
CALL TARRAY (d,count,eoi,status)
```

- INTEGER d or REAL d is the array variable containing the data to be transmitted.
- INTEGER*2 count
  (VALUE) is the number of bytes to be transmitted.
- INTEGER*2 eoi
  (VALUE) is zero if the EOI signal is not desired on the last byte, or one if it is desired.
- INTEGER*2 status
  is a variable which indicates the success or failure of the data transfer.

RARRAY

```
CALL RARRAY (d,count,l,status)
```

- INTEGER d or REAL d
  is the array variable into which data will be received.
- INTEGER*2 count
  (VALUE) is the maximum number of bytes to be received.
- INTEGER*2 l
  is a variable which will be set to the actual number of bytes received.
- INTEGER*2 status
  is a variable which indicates the success or failure of the data transfer.

SRQ

```
LOGICAL*2 SRQ
IF SRQ() ...
```

SETPORT

Note: this routine is not needed on plug-and-play boards like the PCI and Microchannel bus interface cards, since the I/O port address is handled automatically. It needed on the other 488 interfaces only if the I/O address is changed from the standard setting of 2B8 hex.
Note also that you can leave this call out and simply change the configuration file CEC488.INI (see entry on configuration files in the programming chapter).

```
CALL SETPORT (board,ioport)
```

- INTEGER*2 board
  (VALUE) is the IEEE-488 board number (from 0 to 3). Use zero if you have only one IEEE-488 board.
- INTEGER*2 ioport
  (VALUE) is the I/O port address of the IEEE-488 board.

BOARDSELECT

```
CALL BOARDSELECT (board)
```

- INTEGER*2 board
  (VALUE) is the IEEE-488 board number (from 0 to 3).

DMACHANNEL

```
CALL DMACHANNEL (ch)
```

- INTEGER*2 ch
  (VALUE) is the DMA (direct memory access) channel to be used by the 488 interface board. If -1 is used, DMA is disabled for the current board. The default setting is -1 (disabled).

SETTIMEOUT

```
CALL SETTIMEOUT (msec)
```

- INTEGER*2 msec
  (VALUE) is the desired timeout period in milliseconds.

SETOUTPUTEOS

```
CALL SETOUTPUTEOS (eos1,eos2)
```

- INTEGER*2 eos1,eos2
  (VALUE) are the desired end-of-string characters to be appended when the SEND call is used. If only one end-of-string character is desired, set eos2 to zero.

SETINPUTEOS

```
CALL SETINPUTEOS (eos)
```

- INTEGER*2 eos
  (VALUE) is the desired end-of-string character which will cause the ENTER and RECEIVE routines to terminate.

LISTENER PRESENT

```
LOGICAL*2 LISTENERPRESENT
IF LISTENERPRESENT(addr) THEN ...
```

- INTEGER*2 addr
  (VALUE) is the address to be tested.

GPIB BOARD PRESENT

```
INTEGER*2 GPIBBOARDPRESENT
IF (GPIBBOARDPRESENT().EQ.0) THEN ...
```

ENABLE 488EX

```
CALL EN488EX (e)
```

- LOGICAL*2 e
  (VALUE) indicates whether to allow 488EX enhancements.

Examples

Using INITIALIZE, SEND, and ENTER for a simple measurement: (code for **Microsoft FORTRAN** shown)

```
      INTEGER*2 status,l
      CHARACTER*80 r

      CALL INITIALIZE (21,0)
      CALL SEND (16,'F0R0X'C,status)
      CALL ENTER (r,80,l,16,status)
      WRITE (*,*) 'Received length is ',l
      WRITE (*,*) 'Received data is ''',r,''''
      END
```

Using TRANSMIT and RARRAY to receive binary data: (code for Microsoft FORTRAN shown)

```
      INTEGER*2 status,l
      INTEGER*2 d(1000)

      CALL INITIALIZE (21,0)
C     send a command to the device
      CALL TRANSMIT (
  1     'REN MTA LISTEN 3 DATA ''READ'' END'C,status)
C     set device to talk and read the data
      CALL TRANSMIT ('MLA TALK 3'C,status)
      CALL RARRAY (d,2000,l,status)
      END
```

Assembler and other language interfacing

Your board comes with support for many popular programming languages. If you wish to interface directly with the 488 software library from Macro Assembler code or from an unsupported language, this section describes the necessary steps.

Your 488 software is provided as a Microsoft format library in file IEEE488.LIB **(for DOS only, not for Windows)**. This library can be linked with most languages. If you are trying to interface to a language which does not use the Microsoft LINK procedure, you may be able to directly call the on-board firmware, just as in the BASIC language. Direct ROM calls are documented in the file \DOC\ROMCALL.DOC on the applications disk.

If you wish to interface some other **Windows** programming language, you can use the existing Windows DLL libraries. The existing support files for Visual BASIC contain all the declarations and calling information you will need.

Interfacing to some languages will require that you write additional assembler code to handle differences between that language's variable storage and calling conventions and those required by the IEEE-488 library. This is what we have done for languages like QuickBASIC and FORTRAN.

Most language manuals have a section titled something like "interfacing assembly language routines". You will need to read this section.

Interface files

The only file you will need is:

IEEE488.LIB

Compiling programs

The procedure to build an executable program will differ depending on your programming language. With the Microsoft Macro Assembler, simply use the commands:

```
C> MASM myprog;
C> LINK myprog,,,ieee488;
```

The following code shows the assembly language calling sequence for each IEEE-488 interface subroutine.

INITIALIZE

```
extrn ieee488_initialize:far
mov  ax,my_address
push ax
mov  ax,level
push ax
call ieee488_initialize
```

- "my_address" is a value from 0 to 30 indicating the GPIB address for the board.
- "level" is zero to be system controller, two for device mode.

SEND

```
extrn ieee488_send:far
mov  ax,address
push ax
mov  ax,OFFSET string  ; string address
mov  bx,SEG string
push bx
push ax
mov  ax,stringlen
push ax
mov  ax,OFFSET status
mov  bx,SEG status
push bx
push ax
call ieee488_send
```

- "address" if the GPIB address of the device.
- "string" is a character string in memory containing the data to be sent.
- "stringlen" is the length of the string to be sent, in bytes.
- "status" is a word variable in memory which indicates the success or failure of the data transfer.

ENTER

```
extrn ieee488_enter:far
mov  ax,OFFSET rstring    ; string address
mov  bx,SEG rstring
push bx
push ax
mov  ax,maxlen
push ax
mov  ax,OFFSET l
mov  bx,SEG l
push bx
push ax
mov  ax,addr
push ax
mov  ax,OFFSET status
mov  bx,SEG status
push bx
push ax
call ieee488_enter
```

- "rstring" is the string into which the received data will be placed.
- "maxlen" is the maximum number of characters desired.
- "l" is a variable which will be set to the actual received length.
- "addr" is the IEEE-488 address of the device to read from.
- "status" is a variable which indicates the success or failure of the data transfer.

SPOLL

```
extrn ieee488_spoll:far
mov  ax,addr
push ax
mov  ax,OFFSET poll
mov  bx,SEG poll
push bx
push ax
mov  ax,OFFSET status
mov  bx,SEG status
push bx
push ax
call ieee488_spoll
```

- "addr" is an integer indicating the IEEE-488 address of the device to serial poll.

- "poll" is a word variable which will be set to the poll result.
- "status" is a variable which indicates the success or failure of the data transfer.

PPOLL

```
extrn ieee488_ppoll:far
mov  ax,OFFSET poll
mov  bx,SEG poll
push bx
push ax
call ieee488_ppoll
```

- "poll" is a word variable which will be set to the result of the parallel poll operation.

TRANSMIT

```
extrn ieee488_transmit:far
mov  ax,OFFSET cmdstring
mov  bx,SEG cmdstring
push bx
push ax
mov  ax,stringlen
push ax
mov  ax,OFFSET status
mov  bx,SEG status
push bx
push ax
call ieee488_transmit
```

- "cmdstring" is a string containing a sequence of IEEE-488 commands and data.
- "stringlen" is the length of the command string, in bytes.
- "status" is a variable which indicates the success or failure of the data transfer.

```
extrn ieee488_receive:far
mov  ax,OFFSET rstring
mov  bx,SEG rstring
push bx
push ax
mov  ax,maxlen
push ax
mov  ax,OFFSET l
mov  bx,SEG l
push bx
push ax
mov  ax,OFFSET status
mov  bx,SEG status
push bx
push ax
call ieee488_receive
```

- "rstring" is the string into which the received data will be placed.
- "maxlen" is the maximum number of characters desired.
- "l" is a variable which will be set to the actual received length.
- "status" is a variable which indicates the success or failure of the data transfer.

```
extrn ieee488_tarray:far
mov  ax,OFFSET d
mov  bx,SEG d
push bx
push ax
mov  ax,count
push ax
mov  ax,eoi
push ax
mov  ax,OFFSET status
mov  bx,SEG status
push bx
push ax
call ieee488_tarray
```

- "d" is the variable containing the data to be transmitted.
- "count" is the number of bytes to be transmitted.
- "eoi" is zero if the EOI signal is not desired on the last byte, or one if it is desired.
- "status" is a variable which indicates the success or failure of the data transfer.

RARRAY

```
extrn ieee488_rarray:far
mov  ax,OFFSET d
mov  bx,SEG d
push bx
push ax
mov  ax,count
push ax
mov  ax,OFFSET l
mov  bx,SEG l
push bx
push ax
mov  ax,OFFSET status
mov  bx,SEG status
push bx
push ax
call ieee488_rarray
```

- "d" is the variable into which data will be received.
- "count" is the maximum number of bytes to be received.
- "l" is a variable which will be set to the actual number of bytes received.
- "status" is a variable which indicates the success or failure of the data transfer.

SRQ

```
extrn ieee488_srq:far
call ieee488_srq
cmp  ax,0
jz    nosrq
```

SETPORT

```
extrn ieee488_setport:far
mov  ax,board
push ax
mov  ax,ioport
push ax
call ieee488_setport
```

- "board" is the IEEE-488 board number (from 0 to 3). Use zero if you have only one IEEE-488 board.
- "ioport" is the I/O port address of the IEEE-488 board.

BOARDSELECT

```
extrn ieee488_boardselect:far
mov  ax,board
push ax
call ieee488_boardselect
```

- "board" is the IEEE-488 board number (from 0 to 3).

DMACHANNEL

```
extrn ieee488_dmachannel:far
mov  ax,channel
push ax
call ieee488_dmachannel
```

- "channel" is the DMA (direct memory access) channel to be used by the 488 interface board. If -1 is used, DMA is disabled for the current board. The default setting is -1 (disabled).

SETTIMEOUT

```
extrn ieee488_settimeout:far
mov  ax,msec
push ax
call ieee488_settimeout
```

- "msec" is the desired timeout period in milliseconds.

SETOUTPUTEOS

```
extrn ieee488_setoutputeos:far
mov  al,eos1
push ax
mov  al,eos2
push ax
call ieee488_setoutputeos
```

- "eos1" and "eos2" are the desired end-of-string characters to be appended when the SEND call is used. If only one end-of-string character is desired, set eos2 to zero.

SETINPUTEOS

```
extrn ieee488_setinputeos:far
mov  al,eos
```

```
push ax
call ieee488_setinputeos
```

- "eos" is the desired end-of-string character which will cause the ENTER and RECEIVE routines to terminate.

Examples

Using INITIALIZE, SEND, and ENTER for a simple measurement:

```
DATA segment public 'DATA'
  status    dw ?
  len       dw ?
  string    db 'F0R0X'
  stringlen equ $-string
  rstring   db 80 dup (?)
DATA ends

      extrn ieee488_initialize:far
      extrn ieee488_send:far
      extrn ieee488_enter:far
CODE segment public 'CODE'
      assume cs:CODE,ds:DATA
      mov  ax,DATA
      mov  ds,ax
      mov  ax,21              ; initialize (21,0)
      push ax
      mov  ax,0
      push ax
      call ieee488_initialize
      mov  ax,address         ; send
      push ax
      mov  ax,OFFSET string
      mov  bx,SEG string
      push bx
      push ax
      mov  ax,stringlen
      push ax
      mov  ax,OFFSET status
      mov  bx,SEG status
      push bx
      push ax
      call ieee488_send
```

(... continued ...)

```
      mov  ax,OFFSET rstring  ; enter
      mov  bx,SEG rstring
      push bx
      push ax
```

```
      mov  ax,80
      push ax
      mov  ax,OFFSET len
      mov  bx,SEG len
      push bx
      push ax
      mov  ax,16
      push ax
      mov  ax,OFFSET status
      mov  bx,SEG status
      push bx
      push ax
      call ieee488_enter
      mov  ax,4C00H           ; exit to DOS
      int  21H
CODE ends
      end start
```

Your board includes a library of interface routines which work with the IBM OS/2 operating system.

These files from the application disk are used for OS/2 support:

OS2\IEEEOS2.DLL
OS2\IEEEOS2.LIB

The file IEEEOS2.DLL must be copied onto your OS/2 system and placed in a directory which is listed in the LIBPATH command in your CONFIG.SYS file. The \OS2 directory is usually a good place to put this file.

You will also need to make sure that your OS/2 system includes the line "IOPL=YES" in its CONFIG.SYS file.

The file IEEEOS2.LIB should be linked with your program.

Microsoft C (16-bit programs)

Programs written in Microsoft C use exactly the same calls in OS/2 as they do in DOS. A C source program simply needs to be relinked to run in OS/2 mode with the library IEEEOS2.LIB instead of IEEE488.LIB.

IBM C-Set/2 (32-bit programs)

You must follow these additional steps:

1. Edit the file IEEE-C.H and add the keywords **_Far16 _pascal** to all external subroutine declarations.
2. Change all **int** to **SHORT** and all **unsigned** to **USHORT**.
3. Use the **/Gt** compiler option.

Troubleshooting

This section describes solutions to various problems that can occur when conflicts exist among hardware devices in the PC, or among software drivers.

For information on hardware switch settings, see section J on Hardware Configuration.

Checklist for Solving 488 Programming Problems

- ✓ Try running TEST488. If it reports a board problem, check the switch settings and check for conflicts with another add-in card.
- ✓ Swap components if possible. If you have two 488 cards, two cables, and/or two instruments, try swapping them to see if any one component is faulty.
- ✓ IF some device commands work, but not others, it is **NOT** a 488 board problem. It is almost certainly something wrong with the command or data format, possibly the wrong terminating character(s).
- ✓ IF a SEND call returns status 0, but nothing happens on the device, THEN the device has accepted all the characters - therefore, the problem is in the format of the command. Either the command is incorrect (a typo), or the device is still waiting for a terminator. Try adding a return character to the string, or possibly a semicolon.
- ✓ IF the error happens repeatably, but not every time, it is probably also a format error. For example, if ENTER returns the right data once, then returns a null string, then the right data, and so on... this means that the first ENTER call did not read all the characters sent by the device. Changing the input terminator might help.
- ✓ IF the problem looks like a question of programming the correct device command sequence, look for coding examples in the device's manual. Even if the examples are written for a computer like the HP-85, they may be useful.

☞ **Computer fails to boot correctly.**

There is a conflict between the 488 board and other hardware in your PC. **This should NOT happen with plug-and-play boards like the PCI bus interface card.** It may be:

- A memory conflict, usually with a display adapter, network adapter, or expanded memory board. Try a different memory address setting, such as location D000 or C800. You can also disable the ROM memory on 488EX (16-bit ISA bus card). PCI488 and PS488 cannot have a memory conflict.
- A DMA conflict, usually with a network adapter or data acquisition adapter. Try disabling DMA. On PC488 and 488EX this is done by removing jumpers. PCI488 and PS488 cannot have a DMA conflict. If disabling DMA works, you can try using a different DMA channel.
- An I/O conflict. This is very unusual. If there is an I/O conflict, choose another I/O address, such as 2A8, and see the section on CEC488.INI files in chapter 3.

☞ **The GPIB device does not respond.**
(Check the status returned by the IEEE-488 subroutine)

This can be an incorrect device address

- Check the device address switches. You may also want to try the board with a different GPIB device if you have one available.

Or a programming problem

- Try the same SEND and/or ENTER operations with the TRTEST program provided with the board. If these work, double-check your code and look again at the example programs in the language interface sections (A-F) of the manual.

Or a device problem

- Try the board with a different GPIB device if you have one available.

Or a cable problem

- Swap cables if you have another cable available. Inspect the cable and the connectors on the computer and the device for bent pins.

Or a board problem

- Check the board switch settings. Check that the board is firmly seated in the computer's add-in slot. You may want to try the board in another slot in the computer. Also, try cleaning the edge connector of the board with a regular pencil eraser to make sure a good electrical contact is

obtained. Run the TEST488 program to see if the board is OK. If TEST488 is OK, there could still possibly be a problem in the final output stages of the board. Make sure you have tried the other possibilities listed above before resorting to factory repair.

☞Computer hangs when a BASIC program is run.

BASIC488.EXE is not loaded. This program should be run before any BASICA, GWBASIC, or QBASIC program. The BASIC488 command can be put into AUTOEXEC.BAT so it will load automatically when the computer boots.

Or the BASIC program probably has the wrong address setting.

- Very old BASIC programs may have a line like "DEF SEG=&HCC00". This line must match the memory address set in hardware.

☞"Undefined subroutine" in QuickBASIC.

Caused by not loading the IEEE488 library. Use the "/L ieeeqb.qlb" option on the QuickBASIC 4.x command line.

☞"Unresolved external: IEEE488_..." when linking.

You must link the IEEE488.LIB file (for DOS), or WIN488.LIB (for 16-bit Windows), or IEEE_32M.LIB (for 32-bit Windows), which contains these subroutines.

☞"Unresolved external: initialize..." in C.

You MUST include the file IEEE-C.H in your source code. Do NOT edit this file.

☞"Invalid PUBLIC declaration" in Turbo Pascal.

We no longer support Turbo Pascal versions earlier than v7 - you must obtain an older version of the '488 support disk, or get a more recent version of Turbo Pascal.

☞ **488 ROM not found, or not accessible Note: the ROM is not needed for most programming languages. However, it may be used with some very old programs, and some languages, such as Turbo BASIC.**

This can be caused by a hardware memory conflict. Try a different memory address setting, such as location D000 or C800.

On 488EX, make sure the ROM is not disabled (switch S1-7 should be ON to enable the ROM).

Note that newer versions of the boards have no ROM. The older ROM-based boards are now obsolete, and it is strongly recommended that you update your software. This is quite easy, and technical assistance is available. A very limited number of older ROM-based boards will be available for some period of time - contact the factory for details.

If you are running a 386 memory manager such as QEMM-386 or 386Max, it may be hiding the 488 ROM. These memory manager software packages have command-line parameters that allow you to exclude regions of memory. For example, with QEMM, use "X=CC00-CDFF" to exclude the CC00 segment of memory.

There are five models of IEEE-488 interface: PC488 (8-bit ISA), 4x488, PS488 (MicroChannel), 488EX (16-bit ISA), and PCI488 (PCI bus). 4x488 is an obsolete multi-function board. 4x488's configuration and installation is covered in a separate manual. The hardware configuration options of the other boards are covered in the following pages of this Section.

Compatibility

PC488 operates in any IBM PC, XT, AT, the IBM PS/2 model 30 and below, or any PC compatible, with any Intel-compatible processor from 8086 through Pentiums, at any CPU speed. It has been in use since 1983 in thousands of different PC's worldwide.

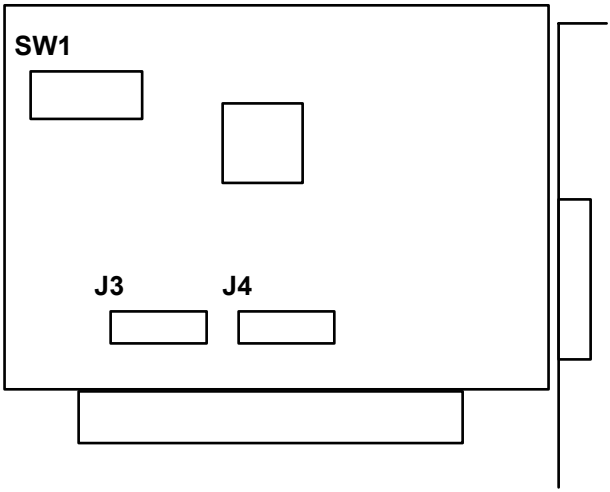
PS488 operates in any Micro Channel computer, which includes the IBM PS/2 model 50 and above, as well as compatibles like the Tandy 5000MC.

488EX operates in any IBM PC/AT or compatible, including 386 and 486 and Pentium processor computers. It requires a 16-bit ISA full-length add-in slot.

PCI488 is compatible with any PCI-bus slot. It is fully plug-and-play compatible.

(also resold as KPC-488.2)

Note that PC488 revision C and earlier has a different layout.
See the next section for this hardware revision!



There is no ROM on this board, unlike the earlier versions of PC488.
The ROM has not been used by our software drivers since 1989, and is now obsolete.

The system controller function of this board is set by the software - no switches are required.

I/O Address Switch (SW1)

Switch SW1 on PC488 controls the I/O address for the GPIB interface chip. This switch is set at the factory to work with most PC configurations. If you do need to change the setting, see the SETPORT subroutine and the section on configuration files in chapter 3 of the manual.

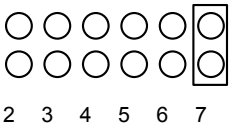
Some options for switch SW1 are shown below:

Position								I/O Address
1	2	3	4	5	6	7	8	
off	on	on	on	on	on	off	off	208
off	on	on	on	on	off	off	off	218
off	on	on	on	off	on	off	off	228
off	on	on	on	off	off	off	off	238
off	on	on	off	on	on	off	off	248
off	on	on	off	on	off	off	off	258
off	on	on	off	off	on	off	off	268
off	on	off	on	on	on	off	off	288
off	on	off	on	on	off	off	off	298
off	on	off	on	off	on	off	off	2A8
off	on	off	on	off	off	off	off	2B8 - default

Interrupt level (J3)

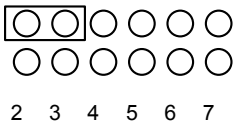
Jumper block J3 controls the interrupt level used by PC488. Most applications do not require interrupts. PC488 is shipped from the factory with interrupts disabled.

You may set the interrupt jumper as shown below:



J3

The default factory setting of the interrupt jumper (disabled) is shown below:



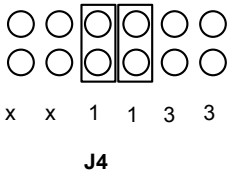
J3

DMA channel (J4)

Jumper block J4 controls the DMA (direct memory access channel) used by PC488. DMA is used for high speed data transfers (see the DMACHANNEL subroutine in Section 3 of the manual).

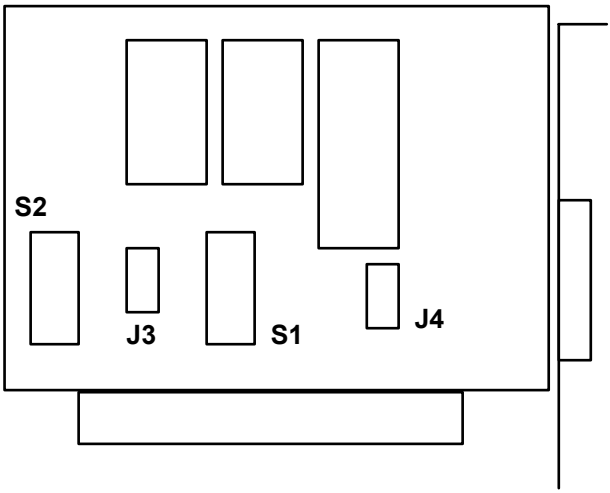
PC488 is configured at the factory to use DMA channel 1. This channel is usually available. If you have a local-area network board, it may use channel 1 and you will need to change PC488's channel or disable DMA on PC488. DMA is not required - it simply makes faster transfers possible.

The factory default DMA settings are shown below:



Note that two jumpers must be installed to select the DMA channel. The example shows jumpers installed for DMA channel 1.

Note that PC488 revision D and later has a different layout. See the preceding section for this hardware revision!



Note: there is an older (pre-1986) hardware version of PC488 which uses the Texas Instruments 9914 interface chip. This version has the part number 01000-00200 on the board. It is not compatible with the newer software described in this manual (except for the direct ROM calls shown for BASIC). If you have one of these cards, call and request an older manual and software disk to go with the TI card.

PC488 has a socket labelled U5 which can accept an optional cache RAM. The cache RAM may be used as a destination for DMA transfers, instead of using a location in the first 640K of system memory. To do this, you must pass an absolute memory address for the cache RAM location to the IEEE-488 subroutine (tarray or rarray). For the majority of IEEE-488 applications, the cache RAM is not required.

Memory Address Switch (S1)

Switch S1 on PC488 controls the memory address for the on-board firmware. This switch is set at the factory to work with most PC configurations. If you find that your computer does not boot correctly with PC488 installed, or that PC488 and another card interfere with each other, you may have to change the switch setting.

S1 Position						Memory Address
1	2	3	4	5	6	
off	off	on	on	on	on	C000 (768K to 784K)
off	off	on	on	on	off	C400 (784K to 800K)
off	off	on	on	off	on	C800 (800K to 816K)
off	off	on	on	off	off	CC00 (816K to 832K)
off	off	on	off	on	on	D000 (832K to 848K)
off	off	on	off	on	off	D400 (848K to 864K)
off	off	on	off	off	on	D800 (864K to 880K)
off	off	on	off	off	off	DC00 (880K to 896K)
off	off	off	on	on	on	E000 (896K to 912K)
off	off	off	on	on	off	E400 (912K to 928K)
off	off	off	on	off	on	E800 (928K to 944K)
off	off	off	on	off	off	EC00 (944K to 960K)

The factory default setting is CC00.

Position 7 of S1 is **not** used.

Position 8 on switch S1 indicates whether or not the PC488 is system controller:

S1 Position 8	
off	system controller
on	device

I/O Address Switch (S2)

Switch S2 on PC488 controls the I/O address for the GPIB interface chip. This switch is set at the factory to work with most PC configurations. If you do need to change the setting, see the SETPORT subroutine and the section on configuration files in chapter 3 of the manual.

Some options for switch S2 are shown below:

Position								I/O Address
1	2	3	4	5	6	7	8	
off	on	on	on	on	on	off	off	208
off	on	on	on	on	off	off	off	218
off	on	on	on	off	on	off	off	228
off	on	on	on	off	off	off	off	238
off	on	on	off	on	on	off	off	248
off	on	on	off	on	off	off	off	258
off	on	on	off	off	on	off	off	268
off	on	off	on	on	on	off	off	288
off	on	off	on	on	off	off	off	298
off	on	off	on	off	on	off	off	2A8
off	on	off	on	off	off	off	off	2B8 - default

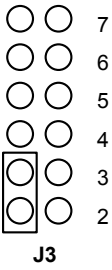
Interrupt level (J3)

Jumper block J3 controls the interrupt level used by PC488. Most applications do not require interrupts. PC488 is shipped from the factory with interrupts disabled.

You may set the interrupt jumper as shown below:



The default factory setting of the interrupt jumper (disabled) is shown below:

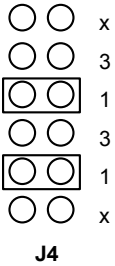


DMA channel (J4)

Jumper block J4 controls the DMA (direct memory access channel) used by PC488. DMA is used for high speed data transfers (see the DMACHANNEL subroutine in Section 3 of the manual).

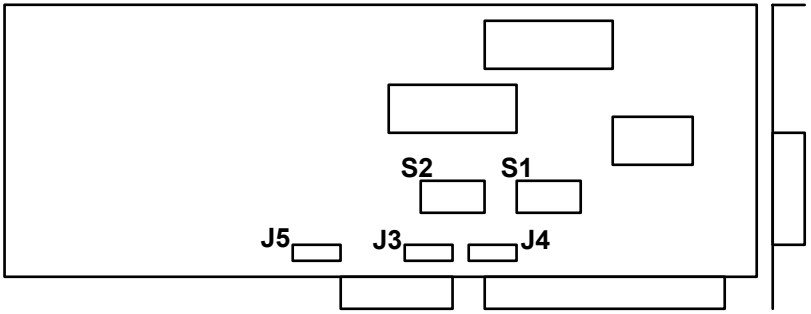
PC488 is configured at the factory to use DMA channel 1. This channel is usually available. If you have a local-area network board, it may use channel 1 and you will need to change PC488's channel or disable DMA on PC488. DMA is not required - it simply makes faster transfers possible.

The factory default DMA settings are shown below:



Note that two jumpers must be installed to select the DMA channel. The example shows jumpers installed for DMA channel 1.

(also resold as KPC-488.2AT)



Memory Address Switch (S1)

Switch S1 on 488EX controls the memory address for the on-board firmware. This ROM is provided for compatibility with PC488 and with some older programs. For most new programs written for 488EX, the ROM is not required and may be disabled. The switch is set at the factory to disable the ROM. If you are running older PC488 programs that need the ROM, find an unused memory region and change the switch setting.

S1 Position							Memory Address
1	2	3	4	5	6	7	
off	off	on	on	on	on	on	C000 (768K to 784K)
off	off	on	on	on	off	on	C400 (784K to 800K)
off	off	on	on	off	on	on	C800 (800K to 816K)
off	off	on	on	off	off	on	CC00 (816K to 832K)
off	off	on	off	on	on	on	D000 (832K to 848K)
off	off	on	off	on	off	on	D400 (848K to 864K)
off	off	on	off	off	on	on	D800 (864K to 880K)
off	off	on	off	off	off	on	DC00 (880K to 896K)
off	off	off	on	on	on	on	E000 (896K to 912K)
off	off	off	on	on	off	on	E400 (912K to 928K)
---	---	---	---	---	---	off	disabled

The factory default setting is **disabled**.

System controller function is set in software on 488EX. Position 8 of S1 is not used.

I/O Address Switch (S2)

Switch S2 on 488EX controls the I/O address for the GPIB interface chip. This switch is set at the factory to work with most PC configurations. If you do need to change the setting, see the SETPORT subroutine in Section 3 of the manual.

Some options for switch S2 are shown below:

```
Position                                  I/O Address
1   2   3   4   5   6   7   8

off on  on  on  on  on  off off      208
off on  on  on  on  off off off      218
off on  on  on  off on  off off      228
off on  on  on  off off off off      238
off on  on  off on  on  off off      248
off on  on  off on  off off off      258
off on  on  off off on  off off      268
off on  off on  on  on  off off      288
off on  off on  on  off off off      298
off on  off on  off on  off off      2A8
off on  off on  off off off off      2B8 - default
```

Note that 488EX uses multiple ranges of I/O addresses. Here are the addresses used when the base address is 2B8:

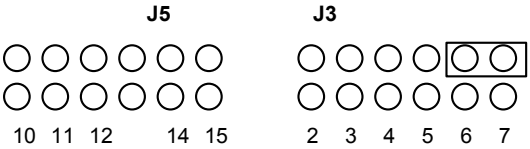
```
 2B8 to  2BF
12B8 to 12BF
22B8 to 22BF
32B8 to 32BF
```

Similar ranges are used for the other base address choices.

Interrupt level (J3/J5)

Jumper blocks J3 and J5 control the interrupt level used by 488EX. Most applications do not require interrupts. 488EX is shipped from the factory with interrupts disabled.

You may set the interrupt jumper as shown below:



Interrupt levels 2 through 7, 10 through 12, 14, and 15 are available. (Level 13 is not available on the PC/AT bus). Select an interrupt level by placing the jumper across a vertical pair of pins on J3 or J5.

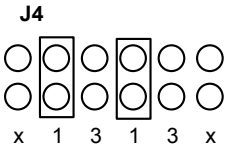
The jumper is shown in its factory default position (disabled).

DMA channel (J4)

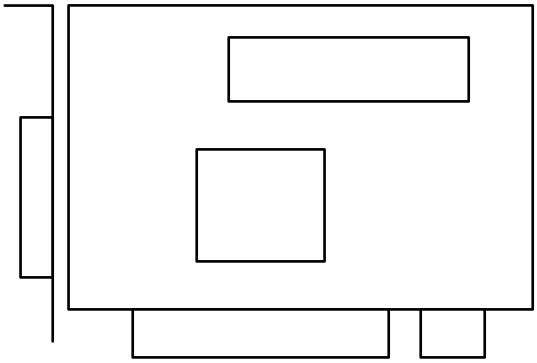
Jumper block J4 controls the DMA (direct memory access channel) used by 488EX. 488EX does not require DMA to acheive high speeds - it includes special accelerator hardware that operates many times faster than DMA rates. However, DMA is provided for backward compatibility with PC488.

488EX is configured at the factory to use DMA channel 1. This channel is usually available. If you have a local-area network board, it may use channel 1 and you will need to change 488EX's channel or disable DMA on 488EX. DMA is not required.

The factory default DMA settings are shown below:



Note that two jumpers must be installed to select the DMA channel. The example shows jumpers installed for DMA channel 1.



PCI488 has no switches or jumpers to set. It is a plug-and-play board, automatically configured by the PCI BIOS functions of the computer.

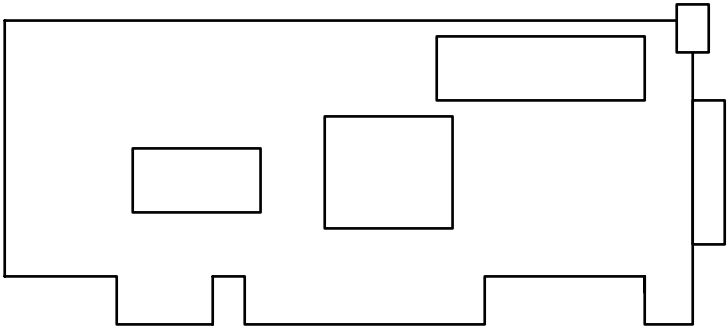
PCI488 uses 8 I/O address registers, and an optional interrupt level. It does not use DMA. It also uses an addtional range of 128 I/O addresses for board configuration.

Forcing a specific I/O address

PCI488 is plug-and-play, so it will configure at whatever I/O address is chosen by the computer at power-up. If you have a very old program written for our '488 cards, **and** you do not have source code (so you cannot simply recompile the program), then you may want to force the card to configure at a known I/O address, such as 2B8 hex (the default setting for our older cards). You can do this with the PCICONF.EXE utility program:

```
C:\CEC488>  pciconf  2B8
```

PS488 (Microchannel, PS/2 board)



PS488 is automatically configured when it is installed, using the features of the IBM Micro Channel.

If you want to change the I/O address, memory address, interrupt level, or DMA channel on PS488, you should boot your computer from the Reference diskette and choose "Set configuration", then "Change configuration". PS488 will appear in the list of installed boards, along with its current settings. You can change these settings as instructed by the Reference diskette configuration program.

The Reference diskette is also useful when you just want to know what the current hardware settings are. Again, boot the computer from the Rereference diskette and choose "Set configuration", then "View configuration".

The usual settings chosen when you install PS488 (if these settings are not already in use by another adapter) are:

- Memory address = CC00
- I/O address = 2B8
- Interrupt level = 3
- DMA channel = 1

Differences from earlier versions

ROM firmware (pre-1989)

The earliest boards contained only on-board firmware in read-only-memory (ROM), which was called directly at a memory address set by switches on the card. This was the only method of adding subroutines easily to the earliest versions of BASIC for the PC, back in 1983.

If you have an existing program written from this time period, and can get source code, it is best to update it. This may take minor edits to the '488 calls. If you cannot obtain source code, the program should still run, as long as you have a version of our hardware containing the ROM, and you set it to the desired memory address (most often hex CC00), and the default I/O address (hex 2B8).

Note that newer versions of the boards have no ROM. The older ROM-based boards are now obsolete, and it is strongly recommended that you update your software. This is quite easy, and technical assistance is available. A very limited number of older ROM-based boards will be available for some period of time - contact the factory for details.

CEC-488 version 2.xx and 3.xx software (1989 to 1995)

These versions of software did not use the on-board ROM (although it is still on the hardware for backward compatibility). Source code written from these versions is 100% compatible with the latest version of software - you can simply recompile and relink.

Version 3.xx added support for the 488EX (16-bit ISA) board. However, the hardware of this board is also backward compatible, so it was not even required to recompile programs to run on the new board.

CEC-488 version 4.xx software

This version added support for 32-bit Windows programs and Windows NT operating system.
Source code did **not** change.
Again, simply recompiling and relinking programs allowed them to run in new operating system environments.

Current version (v5.xx and up)

The latest version includes support for all current hardware, including PCI bus cards, which are plug-and-play configurable.

The only compatibility issue is for older programs written to assume a fixed I/O address of 2B8 hex. If you have source code and simply recompile and relink, the new program will automatically work with all cards, including plug-and-play PCI. If you do not have source code for such a program, you can force the PCI card to run at I/O address 2B8 using the PCICONF utility program:

```
C:\CEC488>   pciconf   2B8
```

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

ASCII	Hex	Decimal	GPIB
space	20	32	LA0
!	21	33	LA1
"	22	34	LA2
#	23	35	LA3
$	24	36	LA4
%	25	37	LA5
&	26	38	LA6
'	27	39	LA7
(	28	40	LA8
)	29	41	LA9
*	2A	42	LA10
+	2B	43	LA11
,	2C	44	LA12
-	2D	45	LA13
.	2E	46	LA14
/	2F	47	LA15
0	30	48	LA16
1	31	49	LA17
2	32	50	LA18
3	33	51	LA19
4	34	52	LA20
5	35	53	LA21
6	36	54	LA22
7	37	55	LA23
8	38	56	LA24
9	39	57	LA25
:	3A	58	LA26
;	3B	59	LA27
<	3C	60	LA28
=	3D	61	LA29
>	3E	62	LA30
?	3F	63	UNL

ASCII	Hex	Decimal	GPIB
@	40	64	TA0
A	41	65	TA1
B	42	66	TA2
C	43	67	TA3
D	44	68	TA4
E	45	69	TA5
F	46	70	TA6
G	47	71	TA7
H	48	72	TA8
I	49	73	TA9
J	4A	74	TA10
K	4B	75	TA11
L	4C	76	TA12
M	4D	77	TA13
N	4E	78	TA14
O	4F	79	TA15
P	50	80	TA16
Q	51	81	TA17
R	52	82	TA18
S	53	83	TA19
T	54	84	TA20
U	55	85	TA21
V	56	86	TA22
W	57	87	TA23
X	58	88	TA24
Y	59	89	TA25
Z	5A	90	TA26
[	5B	91	TA27
\	5C	92	TA28
]	5D	93	TA29
^	5E	94	TA30
_	5F	95	UNT

ASCII	Hex	Decimal	GPIB
`	60	96	SC0
a	61	97	SC1
b	62	98	SC2
c	63	99	SC3
d	64	100	SC4
e	65	101	SC5
f	66	102	SC6
g	67	103	SC7
h	68	104	SC8
i	69	105	SC9
j	6A	106	SC10
k	6B	107	SC11
l	6C	108	SC12
m	6D	109	SC13
n	6E	110	SC14
o	6F	111	SC15
p	70	112	SC16
q	71	113	SC17
r	72	114	SC18
s	73	115	SC19
t	74	116	SC20
u	75	117	SC21
v	76	118	SC22
w	77	119	SC23
x	78	120	SC24
y	79	121	SC25
z	7A	122	SC26
{	7B	123	SC27
|	7C	124	SC28
}	7D	125	SC29
~	7E	126	SC30
DEL	7F	127	SC31

Using PRINT and INPUT for GPIB control

The simplest approach to controlling GPIB instruments with your board makes use of the standard PRINT and INPUT statements. For example, to initialize a Tektronix DM5010 at GPIB address 20 in BASIC:

```
OPEN "gpib20" FOR OUTPUT AS #1
PRINT #1,"INIT"
```

This method of programming can be used whenever data transfer speed is not an important issue. **It cannot be used in Windows NT.**

To set your board up for use with PRINT and INPUT statements, you should install the DOS device driver "CECGPIB.BIN" from the applications disk. Copy UTILITY\CECGPIB.BIN to the root directory of your main system drive. Then, add the following line to your CONFIG.SYS file:

```
DEVICE=CECGPIB.BIN  21
```

where the value 21 specifies the GPIB address to be used by your board, and can be replaced with any value you like between 0 and 30. You must re-boot your computer to complete the installation process.

Once the CECGPIB driver is installed, you can use the normal file OPEN, PRINT, INPUT, and CLOSE statements for access to any GPIB device. The devices are named "gpib1" through "gpib30".

Here is a complete example program in BASIC, showing the use of the PRINT and INPUT statements:

```
5   ' Example - HP9111A digitizing tablet
6   '
10    OPEN "gpib6" FOR OUTPUT AS #1
20    OPEN "gpib6" FOR INPUT AS #2
30    PRINT #1,"IN;DF;CN"        ' initialize
40  '
50    PRINT #1,"OD"             ' ask for data point
60    INPUT #2,X,Y,P            ' read data point
70    PRINT X,Y,P               ' display on screen
80  GOTO 50
```

Lines 10-20 open the device at GPIB address 6. You need to use two different file numbers in BASIC; one for output and one for input.

In line 30, a command is sent to the digitizing tablet to initialize it.

Line 40 sends a command to the tablet, asking it to transmit a pen position.

Line 60 reads the data point into the computer. Since the HP9111A tablet sends the data as ASCII characters (for example "12,25,1"), BASIC can read this information directly into numeric variables.

Here is the same example, given in Turbo Pascal:

```
{ Example - HP9111A digitizer tablet }

program example (output);
var
      tablet_in : TEXT;
      tablet_out : TEXT;
      x,y,p : integer;
begin
      assign (tablet_out,'gpib6');
      rewrite (tablet_out);
      assign (tablet_in,'gpib6');
      reset (tablet_in);
      writeln (tablet_out,'IN;DF;CN'); { initialize }
      while true do
      begin
            writeln (tablet_out,'OD');
            readln (tablet_in,x,y,p);
            writeln (x,y,p);
      end;
end.
```

Here is the same digitizer example, given in Microsoft C:

```
#include <stdio.h>
main ()
{
      FILE *tablet_in,*tablet_out;
      int x,y,p;

      tablet_out = fopen ("gpib6","w");
      setbuf (tablet_out,NULL);
      /* see NOTE below */

      tablet_in = fopen ("gpib6","r");
      fprintf (tablet_out,"IN;DF;CN\n");
      while (1)
      {
            fprintf (tablet_out,"OD\n");
            fscanf (tablet_in,"%d,%d,%d",&x,&y,&p);
            printf ("%d,%d,%d\n",x,y,p);
      }
}
```

NOTE: when programming in C, you should use the "setbuf" call to turn off buffering for the GPIB output file. This forces C to output the commands as it encounters print statements, rather than waiting until a large number of characters have accumulated.

You can send or receive any data to any GPIB device using the PRINT and INPUT statements. If you need to use some of the more advanced GPIB features, such as serial polling, you can use direct CALLs to your board's firmware, as described in section 3.

Firmware CALLs can be mixed with PRINT and INPUT statements. For example, if you have to send a command to a device at GPIB address 4, then wait for a certain serial poll value, then read from the device, you can use the following code in QBASIC:

```
OPEN "gpib4" FOR OUTPUT AS #1
OPEN "gpib4" FOR INPUT AS #2

  PRINT #1,"MEASURE"   ' send command
Lp:
    CALL SPOLL (4,POLL%,STATUS%)
    IF POLL%<>97 THEN GOTO Lp
  INPUT #2,VOLTAGE ' read value
  PRINT VOLTAGE
  CLOSE
  END
```

When mixing CALLs with PRINT and INPUT statements, you'll want to know a little bit about the way PRINT and INPUT affect the GPIB. PRINT assigns the computer as the talker and the device as a listener, and turns on the remote enable signal. INPUT reverses the roles of talker and listener.

The only thing you must avoid when mixing the two programming methods is re-assigning talkers and listeners in a way that the PRINT/INPUT device driver cannot handle. If, for example, you use a PRINT statement to GPIB address 4, then use the TRANSMIT call to turn off all listeners, then PRINT again, the PRINT driver will not know that the listener has been de-assigned, and will not operate correctly. This problem never arises with the SPOLL call.

A few other technical notes on the PRINT/INPUT style of GPIB programming:

Microsoft FORTRAN will not work with this method, due to the way FORTRAN handles file input/output.

Data transfer rates are limited to around 1300 bytes/second (use firmware calls if you need faster transfers)

Anything you can print to the screen or input from the keyboard, you can also transfer to or from a GPIB device. This is also worth remembering as a way of testing your programs. Simply open the "CON" device (display or keyboard) instead of "GPIBn" and see if the data being sent is what you expect. Some devices are very picky about extra spaces or other formatting details.

HP-style universal language driver

This driver allows you to use **any** programming language at all to access your IEEE-488 interface board. (Note: this method cannot be used from Windows NT for 32-bit windows programs).

Installing the driver

Copy the driver file CECHP.EXE to your hard disk. To install the driver, simply run the command:

```
C>  cechp
```

You can optionally specify the GPIB address to be used by the computer. The default is 21, which is the usual choice for HP desktop computers. To use a different address:

```
C> cechp 20
```

You can also load multiple copies of the driver to support multiple interface boards. Give the GPIB address, I/O address (in hex), and board number when you run the driver. For example, to load two copies of the driver for two boards:

```
C> cechp
C> cechp 20 2A8 2
```

Board numbers run from 1 to 4.

You can put the CECHP command in your AUTOEXEC.BAT file so it will run automatically every time you turn on your computer. Look in your DOS manual for examples of setting up an AUTOEXEC.BAT file.

Using the CECHP driver

Once the driver is loaded, it can be accessed from any programming language with simple PRINT and INPUT commands.

Open the driver just like a file. The driver's name is "IEEE". For example, in BASIC:

```
OPEN "IEEE" FOR OUTPUT AS #1
PRINT #1,"OUTPUT 16;INIT"
```

You can PRINT commands to the driver, and use INPUT to get data back. For example:

```
OPEN "IEEE" FOR OUTPUT AS #1
OPEN "IEEE" FOR INPUT AS #2
PRINT #1,"OUTPUT 12;READ"
PRINT #1,"ENTER 12"
INPUT #2,VOLTAGE
PRINT "The voltage is ";VOLTAGE
CLOSE
```

If you have multiple boards, and have loaded multiple drivers, the driver file names are "IEEE", "IEEE2", "IEEE3", and "IEEE4".

The commands used by the CECHP driver are designed to be very close to those provided in Hewlett-Packard BASIC on HP computers. This makes the job of converting old programs quite easy. This section outlines the few differences imposed by the difference in computers.

Device addresses

HP BASIC uses device numbers of the form 7xx, where the "7" is a selector indicating the HPIB interface, and the remaining part of the number is the actual 488 bus address of the device. CECHP uses just the bus address. For example, the HP BASIC command:

```
OUTPUT 716;INIT
```

becomes the CECHP command:

```
OUTPUT 16;INIT
```

Use of the PRINT statement

With CECHP, you must use the PRINT statement to give the command to the CECHP driver. In HP BASIC, the commands are built into BASIC, so the command can be given directly. A line in HP BASIC like:

```
TRIGGER 705
```

becomes this when using PC BASIC:

```
PRINT #1,"TRIGGER 5"
```

Reading data

On an HP computer with built-in HPIB control, you can use a statement like:

```
ENTER 705;A,B,C
```

to read input. With CECHP, you must PRINT the command to the driver, and then read the result with a separate statement:

```
PRINT #1,"ENTER 5"
INPUT #2,A,B,C
```

Other differences

There are other minor differences, as well as some additional capabilities present in CECHP that are not in HP BASIC. These are covered in detail with the description of each command in the following sections.

The most common commands: OUTPUT, ENTER, REMOTE, TRIGGER, etc. are all identical to HP BASIC.

The CECHP driver commands are given below. Lower case indicates argument values which are described in detail in the text. Square brackets [ ] indicate optional parts of commands.

```
ABORT
```

The ABORT commands stops all 488 bus activity and sends an interface clear to all devices.

```
CLEAR address-list
```

The CLEAR command resets one or more devices to their default conditions. CLEAR can be given with no device addresses (a universal device clear), or with a list of device addresses (selected device clear).

For example "CLEAR" resets all devices, "CLEAR 4 8" resets two specific devices.

```
CMDTERM chars
```

CMDTERM specifies the characters which will terminate commands given to the driver. The default choice is return and line feed (CRLF). This choice is OK for almost any programming language, since most languages send return and line feed both at the end of a PRINT statement. The character can be CR, LF, CRLF or any character between double quotes: "x".

```
DISPLAY off
```

The DISPLAY command controls the display of error messages. When an invalid command is given to the driver or it detects an error during a data transfer, it will normally print the message immediately on the screen. If you wish to avoid interrupting other display output, you can use the command "DISPLAY OFF". You can use "DISPLAY ON" to resume error reporting". See also the "ERRORS" command.

```
DMACHANNEL n
```

DMACHANNEL tells the driver to use direct memory access to obtain faster transfer rates. The channel number specified must match the channel selected when the board was installed. Use "DMACHANNEL -1" if you wish to disable DMA.

```
ENTER address [# count]
```

The ENTER command reads data from a device. The address is optional. If no address is given, you should make sure a device is already set to talk and that the computer is set to listen.

If no count is given, the ENTER command reads data until the input terminator is received, or the EOI signal is received. The input terminator is normally a line feed. See the INTERM command if you wish to change the terminator.

If a count is given, using the number symbol and a numeric count value (e.g.: ENTER 14 # 1000), the specified number of bytes are read as binary data. Input terminates on EOI or the speicifed count. No additional line terminating characters are added to the input data.

```
ERRORS
```

The ERRORS command reads the error message, if any, produced by the most recent command. This can be very useful if error display is off.

```
INTERM char
```

The INTERM command specifies the input terminator used with the ENTER command. The default is line feed (LF). The character can be CR, LF, or any character in double quotes: "x".

```
LOCAL address-list
```

LOCAL puts one or more devices back under control of their front panel keys. If no addresses are given, LOCAL simply turns off the remote enable signal on the 488 bus. If addresses are given, a go-to-local command is sent for the specified devices.

```
LOCAL LOCKOUT
```

LOCAL LOCKOUT disables the front panel controls of all devices, so that only the computer can control them. Note that some devices do not have local lockout capability.

```
OUTPUT address-list [# count] ; data
```

The OUTPUT command sends data to one or more devices. The address-list is optional. If no addresses are given, make sure that the computer is already set up as a talker and that listeners have been assigned (you can use multiple OUTPUT commands in a row with addresses only in the first command).

If no count is given, the data begins after the semi-colon and continues until the command terminating character (usually a return).

If a count is given, any binary data values can be sent. The number of characters specified are sent to the device, and no additonal terminating characters are added.

```
OUTTERM chars
```

OUTTERM specifies the terminating characters to be sent at the end of the OUTPUT command (unless binary transfers with a #count are specified). The characters can be CR, LF, CRLF, or any one or two characters in double quotes.

```
PASS CONTROL address
```

This command passes control of the 488 bus to the given device, which must have controller capability.

```
PPOLL
```

This command carries out a parallel poll and provides the resulting value (0 to 255) as input.

```
PPOLL CONFIGURE address value
```

This command sends a parallel poll configuration value to the given device.

```
PPOLL DISABLE address-list
```

This command disables the parallel poll response of the given devices.

```
PPOLL UNCONFIGURE
```

This command disables all parallel poll responses.

```
REMOTE address-list
```

REMOTE puts one or more devices in remote mode, under control of the computer.

```
RESUME
```

RESUME releases the attention (ATN) signal on the 488 bus, letting any data transfers continue.

```
SEND commands
```

The SEND command lets you send any sequence of 488 commands and data.

See the TRANSMIT routine in the main programming section of the manual for details on the commands you can use in SEND.

```
SPOLL address
```

SPOLL serial polls a device and provides its status information as input (0 to 255).

```
SRQ?
```

The SRQ? command provides either a value of 0 or 1 as input, depending on whether service request (SRQ) has occurred since the last time this command was given.

```
TIMEOUT n
```

This commands sets the 488 timeout period in seconds. Values from 0.05 to 60 can be used.

```
TRIGGER address-list
```

TRIGGER sends a 488 bus trigger command to the given devices. This is used to synchronize operations on multiple devices.

```
VERSION
```

This command reads a string giving copyright and version information about the CECHP driver.

Examples

Here are some example programs in BASIC:

```
10 open "IEEE" for output as #1
20 open "IEEE" for input as #2
30 print #1,"OUTPUT 16;F2R0X"
40 print #1,"ENTER 16"
50 input #2,value
60 print "The value is ";value
70 close
```

```
10 '--- Interactive test program example ---
20 cls
30 open "IEEE" for output as #1
40 open "IEEE" for input as #2
50 ioctl #1,"BREAK"     ' make sure driver is reset
60 '
70 line input "> ",cmd$
80 if cmd$="quit" then system
90 print #1,cmd$
100 if ioctl$(1)<>"1" then 70   ' check for input
110 line input #2,i$
120 print i$
130 goto 70
```

Note: the above program shows how to unconditionally reset the driver, using the IOCTL output statement, and also how to test if any input data is available using the IOCTL input function.

Here is an example in Turbo Pascal:

```
program example;
var
      ieeein,ieeeout : TEXT;
      value : real;
begin
   assign (ieeein,'IEEE'); reset (ieeein);
   assign (ieeeout,'IEEE'); rewrite (ieeeout);

      writeln (ieeeout,'OUTPUT 16;F2R0X');
      writeln (ieeeout,'ENTER 16');
      readln (ieeein,value);
      writeln ('Value is ',value);
end.
```

Here is an example written in C:

```
#include <stdio.h>
main()
{
  FILE *ieee;
  float value;

  ieee = fopen ("IEEE","r");
  setbuf (ieee,NULL);       /* see NOTE below */

  fprintf (ieee,"OUTPUT 16;F2R0X\n");
  fprintf (ieee,"ENTER 16\n");
  fscanf (ieee,"%f",&value);

  printf ("Value is %f\n",value);
  fclose (ieee);
}
```

Note: the C program above uses the setbuf(ieee,NULL) call to turn off file buffering for the driver. If this is not done, C will save the output data until it has a preset number of characters (usually 512), and the action will not occur immediately, as you would expect.

Here is an example in Microsoft FORTRAN. Note that FORTRAN does not output the commands immediately to the device driver, so it is necessary to rewind the file after each command.

```
OPEN (10,FILE='IEEE',STATUS='OLD')
WRITE (10,*) 'OUTPUT 16;F2R0X'
REWIND 10
WRITE (10,*) 'ENTER 16'
REWIND 10
READ (10,*) VALUE
WRITE (*,*) 'Value is ',VALUE
CLOSE (10)
END
```

Popular spreadsheet programs have file I/O commands that can be used with the CECHP driver.

To do this, simply create a spreadsheet macro. For example:

```
{OPEN "ieee",W}
{WRITELN "output 15;F0X"}
{WRITELN "enter 15"}
{READLN D1}
{CLOSE}
```

This macro will open the IEEE-488 driver, send a command to a device, then read from the device into cell D1. For specifics on creating macros and assigning them to keys, refer to your spreadsheet software manual.

NOTE: for Microsoft Excel, you can use the built-in version of BASIC to access the Windows version of our direct subroutine calls, just as in the Visual BASIC product. (See section B).

The ENTER command with the #n option allows binary data to be read from a device, without terminating on a particular character value such as line feed.

However, the file input statement used to bring the data into your program also must be set up correctly for binary data. For example, in BASIC the INPUT statement always stops on a carriage return and ignores null (0) bytes and line feeds. The INPUT$ function in BASIC is correct for binary input.

Here are some examples of binary input in various popular languages:

BASIC:

```
open "IEEE" for output as #1
open "IEEE" for input as #2
print #1,"OUTPUT 8;DATA?"
print #1,"ENTER 8 #100"
r$ = INPUT$(100,#2)          ' read binary data
```

C:

```
#include <stdio.h>
main ()
{
      FILE *ieee;
      char data[100];
      ieee = fopen ("IEEE","rb");  /* open binary */
      setbuf (ieee,NULL);
      fprintf (ieee,"OUTPUT 8;DATA?\n");
      fprintf (ieee,"ENTER 8 #100\n");
      fread (data,1,100,ieee);  /* read binary */

      .. etc..
```

Turbo Pascal:

```
program exampleb;
var
   ieeeout : TEXT;
   ieeein : FILE;   { untyped file }
   d : array [1..100] of byte;
begin
   assign (ieeeout,'IEEE'); rewrite(ieeeout);
   assign (ieeein,'IEEE'); reset(ieeein,1);

      writeln (ieeeout,'OUTPUT 8;DATA?');
      writeln (ieeeout,'ENTER 8 #100');
      blockread (ieeein,d,100);

      .. etc..
```

What is IEEE-488.2?

The IEEE-488.2 standard is a 1987 extension to the original IEEE-488 specification. It clarifies the original, and requires more consistent behavior on the part of '488 devices.

IEEE-488.2 also describes a standard set of controller command sequences which can accomplish all the desired functions in a 488.2-compatible system.

The IEEE-488.2 Subroutines

These routines are provided in addition to the standard routines described in the rest of the manual. The two sets of routines may be used together. The 488.2 subroutines are provided in the most popular programming languages: C and QuickBASIC.

Note that ALL control of 488.2 devices can be accomplished with the standard routines. The 488.2 routines simply provide an alternative that follows the naming conventions of the IEEE-488.2 specification document.

When should the 488.2 routines be used?

It's mostly a matter of personal preference as to the style of the different subroutine calls. All 488.2 devices can be controlled with the standard routines.

However, while some non-488.2 devices will work with the 488.2 routines, some will not. In these cases, the standard routines MUST be used. The majority of existing devices are not 488.2 compatible.

The standard routines are easier to use in most cases. For example, with the 488.2 routines, a device must first be placed in remote mode before it can be controlled. The standard routines do all this in one step with SEND.

Disk Files

QuickBASIC support:

4882\QB\IEEE4882.QLB
4882\QB\IEEE4882.LIB
4882\QB\IEEE4882.BI
4882\QB\IEEE4882.OBJ
QB\IEEEQB.BI

C support:

4882\C\IEEE4882.H
4882\C\IEEE4882.OBJ
C\IEEE-C.H
IEEE488.LIB

Short Examples

QuickBASIC 4.x

```
'$INCLUDE: 'ieeeqb.bi'            ' standard include
'$INCLUDE: 'ieee4882.bi'      ' 488.2 include
DIM addrs%(2)     ' 488.2 address list

addrs%(1)=16 : addrs%(2)=9  ' set up list of addresses
addrs%(3)=LASTADDR%

CALL initialize (21,0)        ' standard routine
CALL DeviceClearN (addrs%())  ' clear all the devices
CALL EnableRemote (16)     ' put device 16 in remote
CALL Send1 (16,"F0R0X",LF%)  ' send to device 16
CALL Send1 (9,"START",EOI%)

Lp:                          ' Loop -------------
  WHILE TestSRQ%  ' While SRQ is true
    CALL FindRQS (addrs%,dev%,poll%) ' Get req. device
    CALL Recv (dev%,r$,80,1%,LF%)     ' Read from it
    PRINT dev%;" sent ";r$
  END WHILE
  ' Do other processing here
GOTO Lp
```

C

```
#include "ieee-c.h"           /* standard include */
#include "ieee4882.h"   /* 488.2 include */
int addrs[] = {16,9,LASTADDR}; /* 488.2 addr list */

main ()
{
      int dev;
      char poll,r[81];
      unsigned l;

 initialize (21,0);          /* standard routine */
 ClearDeviceN (addrs);      /* clear all the devices */
 EnableRemote (16);       /* put device 16 in remote */
 Send1 (16,"F0R0X",'\n');      /* send string */
 Send1 (9,"START",END);
 while (1)
 {
      while (TestSRQ())
      {
         FindRQS (addrs,&dev,&poll); /* get device */
         Recv (dev,r,80,1,'\n');     /* read it*/
      }
 }
}
```

QuickBASIC (not QBASIC)

1) Copy the IEEE4882.QLB, IEEE4882.BI, and IEEE4882.LIB files to your hard disk, in addition to the standard file IEEEQB.BI.

2) When you run QuickBASIC, load the 488.2 library:

```
QB /L IEEE4882
```

3) Put the following include lines in your program:

```
'$INCLUDE: 'ieeeqb.bi'
'$INCLUDE: 'ieee4882.bi'
```

Note: if you need to rebuild the .QLB and .LIB files, you will need the files \QB\IEEE4882.OBJ and \QB\IEEEQB.OBJ:

```
link /Q ieee4882 ieeeqb,ieee4882.qlb,,bqlb45;
```

(The BQLB45.LIB file is provided with QuickBASIC).

C

1) Copy the IEEE4882.H and IEEE4882.OBJ files to your hard disk, as well as the standard files IEEE-C.H and IEEE488.LIB.

2) Include these lines in your source code:

```
#include "ieee-c.h"
#include "ieee4882.h"
```

3) To compile and link your program:

Microsoft C for DOS:

```
cl myprog.c ieee4882.obj /link ieee488.lib
```

Most other C versions (with development environments):

```
(put the files IEEE4882.OBJ and IEEE488.LIB in the project file)
```

488.2 Routines: Addresses

Device addresses are passed as integers. Most '488 devices have only a PRIMARY address, in the range 0 to 30. If a device has a SECONDARY address, the address parameter for the 488.2 routine is formed this way:

primary + (secondary * 256)

(That is, the high byte is the secondary address).

488.2 Routines: Address Lists

Some 488.2 routines take a list of addresses, to allow multiple devices to be accessed with one command. These lists are passed as an array of integers, each element being a single device.

The dimension of the list is one more than the number of devices and the end of the list is marked by putting a special value in the array after all the addresses.

See the earlier example programs.

488.2 Routines: Constants

Some pre-defined constants are provided in the 488.2 code:

QuickBASIC	C	Description
LASTADDR%	LASTADDR	used to mark the end of an address list
MAXADDR%	MAXADDR	the maximum number of addresses in a list
EOI%	END	send EOI with last byte
LF%	'\n'	line feed character
CR%	'\r'	carriage return
BUS%	BUS	bus level reset
MESSAGE%	MESSAGE	message level reset
DEVICE%	DEVICE	device level reset (*RST)

All routines modify a global variable which can be checked to see if the status was OK. In QuickBASIC this variable is Ieee.Status%. In C, it is ieee_status

SendCommand

Send a string of bytes as '488 commands. The count parameter tells how many bytes to send.

```
QuickBASIC:     CALL SendCommand (cmds$,count%)
C:              SendCommand ("\x3F\x24",2);
```

SendSetup

Set up the computer to send data and the given list of devices to listen. Many devices must be put in remote mode (using EnableRemote) before sending to them.

```
QuickBASIC:     CALL SendSetup (addrlist%())
C:              SendSetup (addrlist);
```

SendDataBytes

Send a string of bytes as '488 data. The count parameter tells how many bytes to send. The terminator parameter tells what to put on the end of the string. If the terminator is the EOI constant (EOI% in QB, END in C), then EOI is put on the last byte of the string. If the terminator is a number from 1 to 255, then that character code is sent after the end of the string (for example, 10 is a line feed). Both can be combined by using the OR operator (OR in QB, | in C). SendDataBytes assumes that the device has previously been set up to listen via SendSetup.

```
QuickBASIC:     CALL SendDataBytes (s$,count%,term%)
C:              CALL SendDataBytes (s,count,'\n'|END);
```

Send1

Send a string to a single device. Terminator as described in SendDataBytes. Many devices require a previous call to EnableRemote.

```
QuickBASIC:     CALL Send1 (addr%,s$,term%)
C:              Send1 (4,"abc",'\n');
```

SendN

Send a string to multiple devices, in an address list. Many devices require a previous call to EnableRemote.

QuickBASIC:	CALL SendN (addrlist%(),s$,term%)
C:	SendN (addrlist,"def",END);

RecvSetup

Set up the computer to receive and a given device address to send data.

QuickBASIC:	CALL RecvSetup (addr%)
C:	RecvSetup (addr);

RecvRespMsg

Receive data bytes, up to a given maximum number or a specified terminating character. If the terminator parameter is 1 to 255, reception will stop if that character arrives. If the terminator is 0 or the EOI constant, reception will only stop on the maximum count or on the EOI bus signal. Reception always stops on count or EOI, even if a terminating character is specified. Requires a previous call to RecvSetup.

QuickBASIC:	CALL RecvRespMsg (s$,maxlen%,l%,term%)
C:	RecvRespMsg (rstr,80,&len,','');

Recv

Receives data from a specified device. This is the same as calling RecvSetup then RecvRespMsg.

QuickBASIC:	CALL Recv (8,s$,maxlen%,l%,term%)
C:	Recv (addr,rstr,80,&len,END);

SendIfc

Send a interface clear to the whole '488 bus.

QuickBASIC:	CALL SendIfc
C:	SendIfc();

DeviceClear

Send a "selected device clear" to a given device.

QuickBASIC: CALL DeviceClear(12)
C: DeviceClear (addr);

DeviceClearN

Clear (reset) multiple devices. If the address list has no devices (the first element is LASTADDR), then a universal device clear is used to clear all devices on the 488 bus. Otherwise, the given list of devices are cleared.

QuickBASIC: CALL DeviceClearN(addrlist%())
C: DeviceClearN (addrlist);

EnableLocal

Put a single device into local mode, where its front panel keys are active.

QuickBASIC: CALL EnableLocal (8);
C: EnableLocal (addr);

EnableLocalN

Put a list of devices into local mode. If the address list contains no devices (the first element is LASTADDR), then the '488 bus REN (remote enable) signal is turned off. Otherwise, the list of devices are sent a GTL (go to local) command.

QuickBASIC: CALL EnableLocalN (addrlist%())
C: EnableLocalN (addrlist);

EnableRemote

Put a single device into remote, where it may be controlled by the computer.

QuickBASIC: CALL EnableRemote (addr%)
C: EnableRemote (addr);

EnableRemoteN

Put a list of devices into remote mode.

QuickBASIC: CALL EnableRemoteN (addrlist%())
C: EnableRemoteN (addrlist);

SetRWLS

Put a list of devices into remote with lockout state. In this state, not only are the devices controlled by the computer, but their front panel controls are inoperative. (Not all devices implement this.)

```
QuickBASIC:     CALL SetRWLS (addrlist%())
C:              SetRWLS (addrlist);
```

SendLLO

Send the local lockout command. This will put any devices currently addressed to listen into the lockout state.

```
QuickBASIC:     CALL SendLLO
C:              SendLLO();
```

PassControl

Pass control to a device. The device must have controller capability.

```
QuickBASIC:     CALL PassControl (19)
C:              PassControl (19);
```

PPollConfig

Configure the parallel poll response of a device. Note that not all devices respond to parallel polls, and not all are remotely configurable. The parameters specify the device address, the bit number it should use in a parallel poll response (0 to 7), and the sense of the bit (0 or 1) to indicate a true response.

```
QuickBASIC:     CALL PPollConfig (addr%,bit%,sense%)
C:              PPollConfig (8,3,1);
```

PPollUnconfig

Disable the parallel poll response of a list of devices. If the list contains no devices (the first element is LASTADDR), the a universal PPU command is sent. Otherwise, the devices are sent individual unconfigure commands.

```
QuickBASIC:     CALL PPollUnconfig (addrlist%())
C:              PPollUnconfig (addrlist);
```

ReadStatusByte

Read the serial poll status byte of a device. This is similar to the standard routine spoll().

```
QuickBASIC:     CALL ReadStatusByte (addr%,poll%)
C:              ReadStatusByte (8,&poll);
```

Trigger

Send a '488 trigger command to a single device.

```
QuickBASIC:     CALL Trigger (addr%)
C:              Trigger (addr);
```

TriggerN

Send a '488 trigger command simultaneouly to a list of devices.

```
QuickBASIC:     CALL TriggerN (addrlist%())
C:              TriggerN (addrlist);
```

TestSRQ

Test whether the SRQ signal on the '488 bus is true or false. (Note: this is different from the standard routine srq(), which indicates a new SRQ event, then returns false even if the signal remains true).

```
QuickBASIC:     IF (TestSRQ%) THEN ...
C:              if (TestSRQ()) ...
```

Reset488

Reset the '488 system, on any combination of three levels: BUS, MESSAGE, and DEVICE. A BUS reset is a '488 interface clear (IFC). A MESSAGE reset is a '488 device clear (DCL) command. A DEVICE reset is the data string "*RST" sent to the given list of devices (which must be 488.2 compatible). The level parameter can be any combination of these levels.

```
QuickBASIC:  CALL Reset488 (bus%+message%+device%,addrlist%())
C:           Reset488 (BUS+MESSAGE+DEVICE,addrlist);
```

FindRQS

Find a device which is requesting service (SRQ). The given list of addresses are searched in order and the address and status byte of the first device which is sending SRQ is returned. If no device in the list is sending SRQ, then LASTADDR is returned.

```
QuickBASIC:     CALL FindRQS (addrlist%(),addr%,poll%)
C:              FindRQS (addrlist,&addr,&poll);
```

AllSpoll

Serial poll a list of devices and return all their status bytes in an array. Note: the polllist array must be declared to be at least as long as the addrlist array. (In QB, polllist is a integer array. In C, it is a char array.)

```
QuickBASIC:     CALL AllSpoll (addrlist%(),polllist%())
C:              AllSpoll (addrlist,polllist);
```

FindLstn

This routine, given a list of primary device addresses to search, finds which devices are present on the '488 bus (including secondary addresses, if any). The resultlist array will contain the addresses. The maxresult parameter specifies the length of the resultlist array. NOTE: this routine works only on boards which have the listener detection hardware such as the 16-bit ISA bus board. (See also chapter 3 and the GPIBFeature routine).

```
QuickBASIC:     CALL FindLstn (addrlist%(),resultlist%(),maxresults%)
C:              FindLstn (addrlist,resultlist,maxresults);
```

TestSys

Test the given list of devices, using the 488.2 common command "*TST?". This routine is a function, returning the number of devices which fail. As failing devices are detected, their addresses are added to the failures list, and their response value to the responses list. However, the failures and responses arrays will contain only those devices which have failed, so they do not correspond directly to entries in the addrlist array. The failures and responses arrays must be as large as the addrlist array.

```
QuickBASIC:     n% = TestSys (addrlist%(),failures%(),responses%())
C:              n = TestSys (addrlist,failures,responses);
```

488SD: High Speed Streaming Data Protocol

What is 488SD?

The 488SD protocol:

- Allows data transfers at up to 5M bytes/second.
- Is fully compatible with IEEE-488 (488.1 and 488.2).
- Can work in systems with both 488SD and non-488SD devices. 488SD is an extension of the IEEE-488 protocol for high speed data transfer. SD stands for "streaming data", a transfer method used within some computers, and adapted to the IEEE-488 bus.

The 488EX (16-bit ISA bus full-slot card) interface board includes hardware to support the use of 488SD. 488EX can operate at up to 5M bytes/second, depending on the speed of the computer. On a typical IBM PC/AT compatible, the PC bus limits the date rate to around 2.5M bytes/second.

The complete 488SD specification is available from Capital Equipment Corporation, 900 Middlesex Tpk, Bldg. 2, Billerica, MA 01821.

An IEEE-488 system can include both 488SD and non-488SD devices. The first step in using 488SD is for the controller to determine which devices are capable of handling 488SD. If the programmer knows the characteristics of the devices, of course, this information can be written into the controller's program. The 488SD protocol includes a more general method for determining 488SD capability: the stream query message. To see if a device can handle 488SD, the controller sends it:

*STR?

Note: the device must be 488.2 compatible. If non-488.2 devices are present in the system, they must be assumed to be non-488SD. Devices designed to the 488SD specification will respond with a 0 or a 1 data string, indicating whether or not they support 488SD transfers. Devices which do not have any 488SD capability may treat this query as an error, which the controller can detect by reading the standard 488.2 error status register from the device. The example code later in this section shows exactly how to do this.

Data transfer between 488SD-capable devices may use 488SD. When non-488SD devices are involved, the IEEE-488.1 transfer method must be used.

The controller can put 488SD-capable devices into 488SD mode or 488.1 mode with a command:

*STR n

where "n" is 0 or 1 to disable or enable 488SD mode. The device switches modes after receiving the terminating byte of this command message. The next data transfer involving that device will use the new mode setting. Note: the device has up to 100 msec after receiving the command message to switch modes. The controller must wait at least 100 msec before starting a new data transfer involving that device, and is required to readdress the device before the next transfer. This delay is only required when devices are switched into or out of 488SD mode, so it does not impose a significant penalty on overall system throughput.

Since the controller sets up all data transfers by sending talk and listen addresses, it can put the talker and listeners in the correct mode before starting the data transfer. All the controller has to do is to keep track of the current 488SD mode setting of the 488SD-capable devices, then make sure

to send commands to the talker and listeners to put them into matching modes, if necessary.

For example, if device 1 (which is to be the talker) is 488SD-capable, and is currently in 488SD mode, and device 2 (which is to be the listener) is not 488SD-capable, the controller must send a "*STR 0" message to device 1 before setting up the transfer. To do this, the controller makes itself the talker, and device 1 the listener. The controller then puts itself into 488SD mode, because device 1 is currently in that mode, and sends "*STR 0". Then, the controller can set up the transfer by making device 1 a talker and device 2 a listener.

The 488SD data transfer runs at high speed because it transfers multiple data bytes without waiting for the handshake signals between bytes. Listeners can still stop the incoming data when they need more time, but the overhead of using the three-wire 488.1 handshake on every byte is eliminated. This lets 488SD go 5 times faster than 488.1. 488SD does require the device to have some high speed circuitry which is not very difficult or expensive today, but was not practical in the 1970's when the 488.1 standard was written.

There is one more important 488SD command, which sets the 488SD "timing parameter". 488SD can achieve its maximum data rate only with short cables and a limited number of devices. Physical effects require a slower rate as cable length and number of devices increase. By default, 488SD devices run at a rate which will work on the longest cable and most devices allowed by IEEE-488. To get the higher rates, the programmer must know the system cabling, look up the allowed timing parameter value in a 488SD timing table (see later in this section), and send a command to each 488SD-capable device to tell it to use the faster timing. The command is:

*STT n

where "n" is the 488SD timing parameter value from the table.

Your board's software uses the Enable_488SD routine to control the 488SD capabilities of the 488EX interface board. To set the 488SD mode of external devices, you must send the "*STR n" and "*STT n" messages. To set the 488SD mode of the 488EX card, use this call:

enable_488sd (enable,timing)

where:

- enable is true or false to indicate whether 488SD mode is to be used.
- timing is the 488SD timing parameter value from the table (see table later in this section).

The sections on each programming language show exactly how to call this routine.

488SD mode is used only in the **tarray** and **rarray** routines. Other routines always use the 488.1 protocol. Therefore, when 488SD mode is active, **transmit** must be used to set talkers and listeners, and tarray or rarray used for the data transfers.

Note 1: if the 488EX board is the GPIB controller, and a data transfer is occurring between two other devices (the board is not participating), you should turn 488SD mode off on the board.

Here is sample code to send data to a known 488SD device. It is assumed that the device is not currently in 488SD mode.

```
send (8,"*STR 1",&status);   /* put device in 488SD */
pause(0.1);     /* delay 0.1 second */
enable_488sd (1,250);       /* put 488EX in 488SD */

transmit ("MTA LISTEN 2",&status);
tarray (info,2000,1,&status);    /* send the data */

tarray ("*STR 0",6,1,&status); /* put in 488.1 */
pause(0.1);
enable_488sd (0,250);
```

In this example, the device and controller return to 488.1 mode after the data transfer. If additional transfers were required, there is no need to switch back and forth every time - the devices can be left in 488SD mode.

Here is sample code for determining which devices are 488SD-capable, shown in C:

```
/* this subroutine takes an array of device addresses,
  which are known to be 488.2 devices, and returns
  and array indicating which ones are 488SD-capable */

find_488sd (int Addresses[],int nDevices,int sd[])
{
  int i,status;
  char poll;
  char response[2];
  unsigned len;

  transmit ("IFC",&status);  /* reset all devices */
  for (i=0;i<<nDevices;i++)
  {
      /* clear the device status */
      send (Addresses[i],"*CLS",&status);
      /* enable error status 32 (bad command) */
      send (Addresses[i],"*ESE 32",&status);
      /* send the 488SD query */
      send (Addresses[i],"*STR?",&status);
      /* get device status byte */
      spoll (Addresses[i],&poll,&status);
      if (poll & 0x20) {   /* error */
         sd[i] = 0;        /* not 488SD */
      }
      else {
         /* get response */
         enter (response,2,&len,Addresses[i],&status);
         if (response[0] == '1')
            sd[i] = 1;
         else
            sd[i] = 0;
      }
  }
}
```

The 488SD protocol allows data transmission rates which depend on the cable length in the IEEE-488 system. Here is a list of the available choices (note that some devices may not implement all choices - they will use the next slowest rate that they support):

Cable Length (m.)	488SD Timing (nsec)	488EX timing (nsec)
1	100	100
2	120	200
4	150	200
6	200	200
8	225	300
10	250	300
15	350	400
20	500	500

The 488SD column shows the specification's minimum allowed timing for the given cable length. The 488EX column shows the next larger available choice on the 488EX card. You may call the enable_488sd routine with either value, since the software will round up automatically.

488SD requires that there be one device per 2 meters of cable, to ensure correct cable termination.

4

488EX 3-40, 15-11
488SD 21-1

A

Commands ... 2-8, 3-26, 3-30
AUTOEXEC.BAT . 7-1, 19-1

B

BASIC 8-1
Language 7-1
BASIC Language 7-1
BASIC488 ..... 7-1, 14-4
Binary Data . 3-36, 3-38, 3-39
BoardSelect 3-43, 4-24
Borland C++ 10-1
Borland Delphi 9-1

C

Language 10-1
C Language 10-1
C++ 10-1
Cables 6-9
Cache RAM 15-6
CALL ABSOLUTE 8-12
CEC488.INI 3-47
CECGPIB 18-1
CECHP 19-1
Clear 3-32
CMD 3-29
Configuration 3-47
Error 14-3
Conflicts, resolving 14-3, 15-7
Controller 3-5, 4-2, 4-17

D

DATA 3-23
Data Formats 3-39
Data Transfers 3-6, 3-8, 3-23, 3-36
DCL 3-29, 3-32
DEF SEG Statement 7-1
Delphi 9-1
Address 2-5
Device Clear 3-29, 3-30, 3-32
Device Driver 18-1, 19-1
Device Mode 4-2
Device Trigger 3-33
Direct Memory Access . 3-40
DMA 14-3, 15-5, 15-10
DmaChannel 3-41

E

Electrical specs 6-4
Enable 488SD 21-4
END 3-23
Enter 3-8, 3-19
EOI 3-8, 3-19, 3-26, 3-36
Excel 19-13

F

Feature 3-18
FORTRAN 11-1

G

GET 3-30, 3-33
Address 2-5, 3-5
GPIBFeature 3-18
GTL 3-26

H

Handshake 6-5
Hardware Configuration 15-1
Hardware, direct access .. 4-4
High Speed Transfers ... 3-40
HP-style 19-1

I

Address . 3-42, 14-3, 15-3, 15-8, 15-12
I/O Address . 3-42, 14-3, 15-3, 15-8, 15-12
I/O Port 3-47
IEEE-488 Standard .. 1-2, 6-2
IEEE-488.1 2-4
IEEE-488.2 2-4
IFC 3-31
Initialize 3-5
Interface Clear 3-31
Interrrupts 15-13
Interrupts 4-19, 4-21, 15-4, 15-9

J

Jumper .. 15-4, 15-5, 15-9, 15-10, 15-13

L

FORTRAN 11-1
Lahey FORTRAN 11-1
Error 14-4
LISTEN 3-22
Listener 2-6, 3-22
Listener Present 3-16
LLO 3-29
Local 3-26, 3-29
Lotus 1-2-3 19-13

M

Mechanical specs 6-7
Address 7-1, 14-3, 15-7, 15-11
Memory Address . 14-3, 15-7, 15-11
Memory manager 14-5
FORTRAN 11-1
MLA 3-23
MTA 3-22
Commands 3-22, 3-29
Multiple boards 3-42, 3-43, 4-24

N

NEC 7210 4-4

O

OS/2 13-1

P

Parallel Poll 3-15, 3-28
Language 9-1
Pascal Language 9-1
Passing Control 3-30, 4-18
PC488 ..... 15-2, 15-6
Power BASIC 8-2
PPC 3-28
PPD 3-28
Ppoll 3-15
PPU 3-28
Address 2-5
PRINT and INPUT 18-1, 19-1
Problems 14-1
PS488 15-15

Q

QBASIC 8-1
QEMM 14-5
Quattro 19-13
QuickBASIC 8-2

R

Rarray 3-38, 3-40
Rate 3-40
Receive 3-34
Registers 4-4
Remote 3-26
REN 3-26
RM/FORTRAN 11-1
FORTRAN 11-1

S

SCPI 2-4
SDC 3-30, 3-32
SEC 3-22
Address 2-5, 3-7, 3-22

Secondary Address ..2-5, 3-7, 3-22
Send 3-6, 3-19
Serial Poll 3-14, 3-27
Service Request ... 3-13, 4-16
SetInputEOS 3-46
SetOutputEOS 3-45
SetPort 4-24
SetTimeout..... 3-44
SPD..... 3-27
SPE 3-27
Specifications, Electrical 6-4
Specifications, Mechanical6-7
Speed 3-40
Spoll 3-14
Status .3-6, 3-8, 3-14, 3-20, 3-34, 3-36, 3-38, 14-3
Status bits..... 4-7, 4-17
STR$ (BASIC) 3-25
Switch ... 15-3, 15-7, 15-8, 15-11, 15-12
System controller ... 2-2, 15-7

T

TALK 3-22
Talker..... 2-6, 3-22
Tarray.....3-36, 3-40, 7-4
TCT..... 3-30
Terminator ..3-19, 3-23, 3-34, 3-45, 3-46
Error..... 14-3
Timeout.....3-6, 3-44, 14-3
Transfer rate..... 3-40
Transmit..... 3-20
Trigger 3-30, 3-33, 4-17
Troubleshooting..... 14-1
Turbo BASIC..... 8-2

U

U5 15-6
Commands..... 2-8
Universal language driver. 19-1
UNL..... 3-22
Error..... 14-4
Unresolved external 14-4
UNT 3-22

V

VARPTR (BASIC) 7-4
VB..... 8-1
Visual BASIC 8-1

Artisan Technology Group is an independent supplier of quality pre-owned equipment

Gold-standard solutions

Extend the life of your critical industrial, commercial, and military systems with our superior service and support.

We buy equipment

Planning to upgrade your current equipment? Have surplus equipment taking up shelf space? We'll give it a new home.

Learn more!

Visit us at **artisantg.com** for more info on price quotes, drivers, technical specifications, manuals, and documentation.

Artisan Scientific Corporation dba Artisan Technology Group is not an affiliate, representative, or authorized distributor for any manufacturer listed herein.

We're here to make your life easier. How can we help you today?

(217) 352-9330 | sales@artisantg.com | artisantg.com

ARTISAN® TECHNOLOGY GROUP