cec 488 PROGRAMMING and REFERENCE MANUAL

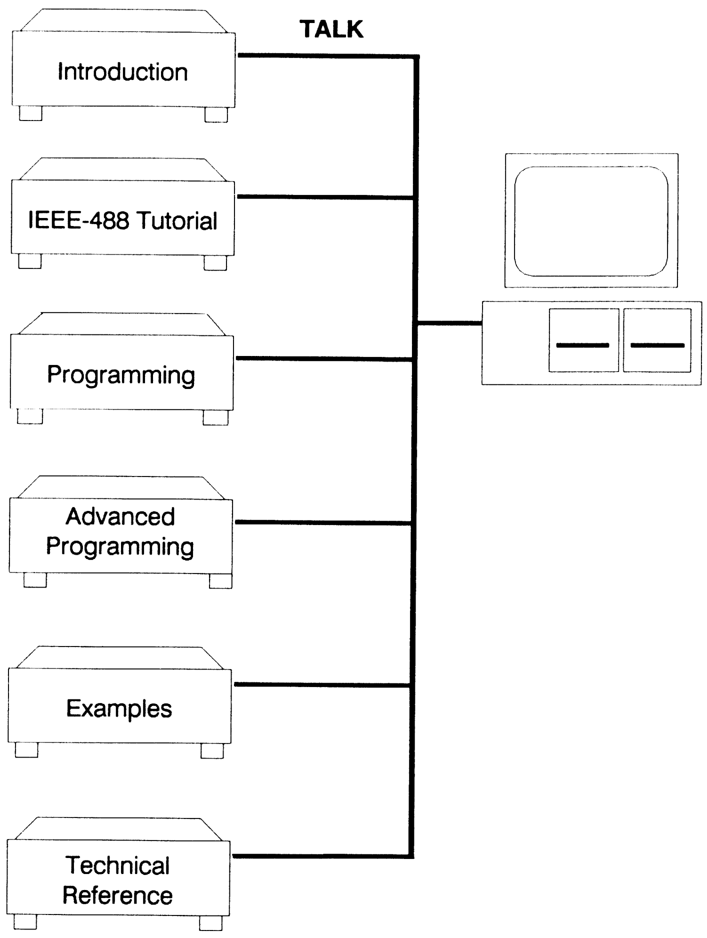
IEEE-488 Instrumentation Interfaces from Capital Equipment Corporation.

Capital Equipment Corporation
99 South Bedford Street #107
Burlington, Massachusetts 01803
(617) 273-1818

CEC IEEE-488 Programming and Reference Manual.
Part number 01000-90310

93 92 91 90
10 9 8 7 6 5

Table of Contents



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)

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

CEC manufactures three models of IEEE-488 interface: PC< >488, 4x488, and PS< >488. Although installation differs for these products, programming is identical. Throughout this manual, the term CEC-488 will be used when the remarks apply to all three products.

CEC-488 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.

CEC-488's key features include:

- Implementation of the entire IEEE-488 standard. CEC-488 can operate as a system controller or a device.
- A choice of programming methods: simply using the PRINT and INPUT statements of your programming language, or using CALLs directly to the firmware routines for the maximum speed and flexibility.
- High-speed direct memory access transfers.
- Support of HPIB printers and plotters for use with MS-DOS, wordprocessing, spreadsheet, and graphics programs.

4x488 also provides:

- Up to 4 megabytes of extended or expanded memory (Lotus-Intel-Microsoft version 4.0 compatible).
- An RS-232 (serial) communications port.
- A parallel printer port.

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

Installation and Checkout

Installation of each of CEC's IEEE-488 interfaces is covered in the sections which follow. Once you have installed your interface card, use the program TEST488 from the application disk to check that it is operating correctly:

```
C> A:TEST488
```

If TEST488 reports any errors, it will suggest possible actions to correct the problem. You may also want to read the appendix on Troubleshooting. If you need to call for technical support, please run the program PCMAP from the application disk and get a printout of the result, as well as a printout of the TEST488 error messages.

After you have tested the board, try running the TRTEST program from the application disk:

```
C> A:TRTEST
```

This program lets you try sending and reading data interactively.

PC< >488 Installation

Remove the PC< >488 interface board from its protective packaging by grasping the metal rear panel. Save the anti-static bubble package since it may be used to return the board if it needs repair.

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

In most cases, there will be no need to change any switch settings on PC< >488.

Set the PC power switch to OFF. Unplug the power cord and disconnect all cables from the rear of the system unit.

Use a screwdriver to remove the cover mounting screws from the rear of the system unit.

Slide the system unit's cover away forward. When the cover will go no further, tilt it up, remove it from the base, and set it aside.

PC< >488 can be installed in any slot in the computer, except slot number 8 in a PC/XT.

Use a screwdriver to remove the screw which secures the rear panel cover on one of the computer's add-in slots. Save the screw for installation of the card.

The GPIB connector should be pressed through the rear panel first, and then the board should be pressed into the main board expansion slot.

Secure PC< >488 with the rear panel mounting screw.

Slide the system unit's cover back on. Align the cover and reinstall the mounting screws.

Turn your PC back on and run the TEST488 program as shown earlier. This will verify correct operation of PC< >488.

If you have any conflicts with other boards in your computer, you will need to change switch settings on PC< >488. In almost every case, the conflict will be in the memory address switch setting (S1). See the Hardware Configuration appendix for information on switch settings.

The PCMAP program provided on the applications disk can be used to determine which memory addresses are empty on your computer. PC< >488 is set at the factory to an address (hex CC00) which is usually empty.

4x488 provides memory, serial port, printer port, and an IEEE-488 port.

The memory on 4x488 adds to your current computer memory in one of three ways: as conventional memory, extended memory, or expanded memory.

Conventional memory

This is memory within the normal DOS 640K limit. If your computer has less than 640K, 4x488 can fill it out to the full DOS capability.

Extended memory

Extended memory is all memory addressed above the 1 megabyte boundary. This memory is not accessible to most DOS programs. It can be used with the DOS VDISK program to provide a RAM disk.

On computers with an 80386 processor, software can make extended memory look like expanded memory. This additional software may come with your computer (for example, Compaq's CEMM program), or it may come with other applications (for example, Quarterdeck's DesqView program).

(Note that there is a gap between the DOS 640K and the beginning of extended memory. This gap contains your video memory and add-in board BIOS memory, like the IEEE-488 firmware ROM on 4x488).

Expanded memory

Expanded memory is a solution to the DOS 640K limitation. Programs which are designed to use expanded memory can store data or program code outside of the usual DOS memory area. There can be up to 32 megabytes of expanded memory in a computer. Expanded memory is organized according to an industry standard called LIM version 4.0, which is supported by software provided with 4x488.

Note that the nature of the LIM expanded memory standard is such that only one expanded memory manager software package can exist at a time, and this software is hardware-dependent, so expanded memory must come from only one manufacturer's add-in boards. You cannot mix expanded memory from AST, for example, with expanded memory on 4x488. You can, however, have expanded memory on one board and extended memory on the other.

Installation

Remove the 4x488 interface board from its protective packaging by grasping the metal rear panel. Save the anti-static bubble package since it may be used to return the board if it needs repair.

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

In most cases, there will be no need to change any switch settings on 4x488.

Find the separate rear panel plate containing the serial and parallel port connectors. Connect the ribbon cables leading from this plate to the top edge of the 4x488 board. The 10 conductor cable from the serial port connector goes to JS on the board. The 25 conductor cable from the parallel port connector goes to JP on the board. The ribbon cables are keyed so they will only attach in the correct orientation.

Turn the computer off. Unplug the computer and disconnect any cables leading from the rear panel.

Unscrew the rear panel screws which hold the cover in place. Most computers have a screw at each corner, and one in the center at the top.

Slide the cover forward until it stops, then tilt it up to completely remove it from the base. Set the cover aside.

Choose an expansion slot you wish to use for 4x488. 4x488 must be installed in a 16-bit expansion slot (with two connectors). The next slot to the right should be unused if you wish to install the serial and parallel port connectors.

Remove the rear panel cover plates from the chosen expansion slots. Save the screws for use later.

Press 4x488 carefully into the expansion slot. Tilt the back end of the board downward to allow the connectors to fit through the rear panel, then lower the front end of the board into place. Make sure it is fully seated into the connectors. Install the parallel and serial port panel into the adjacent slot.

Use the screws saved earlier to secure the board and connector panel into their slots.

Replace the computer cover and attach it with the rear panel screws.

Turn the computer on and run DOS in the normal fashion.

Next, run the 4x488 INSTALL software:

- Hard disk systems:
  - Insert the 4x488 installation disk in drive A.
  - Type "A:", then Enter.
  - At the "A > " prompt, type "INSTALL", then Enter.
- Dual floppy disk systems:
  - Leave your DOS system disk in drive A.
  - Insert the 4x488 installation disk in drive B.
  - Type "B:", then Enter.
  - At the "B > " prompt, type "INSTALL", then Enter.

The INSTALL software will begin by asking you for the disk drive containing the DOS system. On hard disk systems, this is normally drive "C"; on floppy disk systems, this is drive "A".

Continue as prompted by INSTALL. INSTALL will recommend a memory and port configuration for 4x488, and then ask if you wish to make any changes.

Look carefully at the configuration screen. It indicates the recommended settings for 4x488, as well as the other memory and I/O port hardware already present in your computer. If you have a computer with the 80386 processor, INSTALL will recommend that all your memory be configured as extended memory. Many 80386 systems (such as Compaq's) come with software which turns extended memory into expanded memory. When you have such software (Compaq's version is called CEMM), you should accept INSTALL's recommendation, and use the software provided with the computer to get expanded memory if you want it.

The available configuration options are:

- Address of expanded memory page area
  This is the memory location used to access expanded memory. If expanded memory is not used, no paging area is necessary. Possible choices include:
  - none
  - C0000, C4000, C8000, CC000, D0000

- Amount of memory to add to system RAM (FILL)
  This is the amount of conventional memory to be allocated to bring your system up to the full DOS 640K.
- Amount of "extended" memory
  This can be any value from zero to the total memory on your 4x488, in 128K increments. When you have a computer with the 80386 processor, INSTALL recommends that all memory be extended memory.
- Amount of "expanded" memory
  This can be any value from zero to the total memory on your 4x488, in 128K increments.
- IEEE-488 firmware address
  This is the memory location of the IEEE-488 firmware ROM. Possible choices are:
  - C0000, C4000, C8000, CC000, D0000, D4000, D8000, DC000
- Parallel port
  This is the I/O address for the 4x488 parallel printer port. INSTALL enables this port if possible. If you already have a port that conflicts with the available choices, the port is disabled.
  - none
  - 278H
- Serial port
  This is the I/O address for the 4x488 serial port. INSTALL enables this port if possible. If you already have a port that conflicts with the available choices, the port is disabled.
  - none
  - 3F8H (COM1)
  - 2F8H (COM2)

In most cases, you should answer "NO" to indicate that you do not wish to change the recommended settings.

INSTALL will copy the 4x488 software to your system disk and modify your CONFIG.SYS file to run the 4x488 software.

INSTALL will end by reporting the port settings you have chosen. Note these so that you will know which board to connect cables to.

NOTE: Some versions of the IBM serial/parallel card document the port settings backwards. You may think you have chosen COM1 when in fact you have chosen COM2, and 4x488 will become COM1.

NOTE: You should NOT run the SETUP program provided with your computer to change the memory size settings.

Remove the installation disk from the drive and reboot your computer to complete the installation.

When DOS starts up, you should see a message similar to this:

```
CECEMM v2.2 - DOS Driver software for 4x488
   LIM version 4.0 expanded memory
   Copyright (C) 1988, Capital Equipment Corp.

  PC-488 firmware ROM installed at segment CC00 (hex)
  There are now a total of 2 parallel ports available
  There are now a total of 2 serial ports available
  There is now   640K of base memory
  There is now     0K of extended memory
  There is now  2048K of expanded memory
```

If any error messages appear, see the Troubleshooting appendix of this manual. If you have trouble, make sure to run the PCMAP program from the application disk, the EMMTEST program from the installation disk, and write down or print the results.

Remove the PS< >488 interface board from its protective packaging by grasping the metal rear panel. Save the anti-static bubble package since it may be used to return the board if it needs repair.

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

PS< >488 takes full advantage of the Micro Channel's automatic configuration capability, so there are no switches to set. However, the computer will need some information about the board. This information is contained in an option file on the PS< >488 disk.

You will need the following items:

- PS/2 Reference diskette (comes with your computer)
- A blank diskette (if you have not already made a working copy of the Reference diskette)
- The 488 application diskette
- The PS< >488 interface board

Because of the way IBM designed PS/2 automatic configuration to operate, whenever you add any board to the PS/2 the following steps are necessary:

Before putting PS< >488 into the computer:

If you have not already made a working copy of your PS/2 Reference diskette, do so now. To do this, boot the computer from the Reference diskette. When the copyright screen appears, hit Enter to go to the main menu. Select "Back-up the Reference diskette". Follow the computer's instructions, using a blank diskette to make a working copy of the Reference diskette.

Add the PS< >488 option information to the Reference diskette. To do this, put your working copy of the Reference diskette in drive A. Return to the main menu (or reboot the computer). Select "Copy an option diskette". Follow the computer's instructions. The option diskette is the 488 application diskette.

Now, turn the computer off and install PS< >488 in an unused slot.

Remove the back panel cover from an unused slot. Push PS< >488 firmly into the slot. Make sure that the rear panel bracket and plastic board guides align correctly. Tighten the rear panel screw.

Put you working copy of the Reference diskette in drive A and turn the computer on. When the copyright screen appears, hit Enter. A message window should appear (you will need to hit PgDn to read all of it), which will ask if you would like to run automatic configuration. Type "Y". When automatic configuration completes, you can remove the Reference diskette and hit Enter to reboot the computer.

Note: if you wish to run programs written in BASICA or GWBASIC with your PS< >488, you will need to load the software from disk. To do this, run the program PS488 from the application disk:

```
A> PS488
```

You can copy this program to your system disk, and put the PS488 command in your AUTOEXEC.BAT file to make it run automatically.

This program is not necessary if you are using other programming languages.

If you intend to use CEC-488 with GPIB peripherals such as printers and plotters, no programming is necessary. All you need is the interface card and the PRINTER and/or SERIAL utility programs provided on the applications disk. These utilities let you use your favorite wordprocessing or spreadsheet programs with GPIB peripherals.

PRINTER allows you to make GPIB devices look like standard parallel printer ports (LPT1, LPT2 or LPT3). 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 (or PRN), insert the CEC-488 applications disk and type:

```
A> \UTILITY\PRINTER 1
```

Up to three values can follow PRINTER on the command line. These values specify the devices that will appear as LPT1, LPT2, and LPT3. 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. P1 refers to the first installed port, usually referred to as LPT1 or PRN. P2 refers to the second parallel port, and P3 refers to the third parallel port.

Example. Install a regular parallel printer as LPT1 and install your GPIB plotter (at address 5) as LPT2. The command would be:

```
A> \UTILITY\PRINTER P1 5
```

SERIAL allows you to make GPIB devices look like standard serial ports (COM1 or COM2). 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, insert the CEC-488 applications disk and type:

```
A> \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. S1 refers to COM1 and S2 refers to COM2.

Any program which uses the operating system to output characters to a serial port will work with SERIAL and GPIB devices. Some programs, however, bypass the operating system and handle the serial port hardware themselves. SERIAL cannot redirect output from programs that bypass the operating system. If this is the case with your program you may want to try the parallel port output option (using PRINTER) or output to a disk file which can then be output to the device using SERIAL or PRINTER.

To use CEC-488 as a plotter or printer interface with Lotus 1-2- 3, see the documentation file \DOC\LOTUS.DOC on the CEC-488 applications disk. This file can be read by entering "TYPE A:\DOC\LOTUS.DOC" at the DOS prompt.

To use CEC-488 as a plotter interface with Microsoft Windows, see the documentation file \DOC\WINDOWS.DOC on the CEC-488 applications disk. This file can be read by entering "TYPE A:\DOC\WINDOWS.DOC" at the DOS prompt.

PRINTER and SERIAL may be used with instruments as well as printers and plotters. Installing a device to look like a standard parallel port means that you can write data to the device using the built-in line printer statements of your programming language.

PRINTER and SERIAL are executable files which means they can be included in your AUTOEXEC.BAT file for automatic installation on power-up. If you need information on creating an AUTOEXEC.BAT please refer to your DOS manual.

Using CEC-488 with Third Party Software

CEC-488 is compatible with a number of software packages for instrument control and data acquisition available from other vendors. These include ASYST and ASYSTANT-GPIB from MacMillan Software, EasyWave from LeCroy, TBASIC or HTBASIC from TransEra, and others.

All these programs include their own handling of the CEC-488 interface, so no additional programming is required.

For some of these programs, you will need to know the I/O address of the CEC-488 interface. This is usually set to 2B8 hexadecimal.

Using CEC-488 for Programmed Instrument Control

To use CEC-488 for custom data acquisition, instrument control, or testing, you will want to write specialized software. CEC-488 has been designed to make this easy, by providing a simple set of high-level routines that may be used with all of the popular programming languages.

Our companion software product, **Co-Operator™**, can make programming even easier. Co-Operator provides interactive menus and help functions, application templates, and instrument libraries to make your job easier. Co-Operator can generate executable programs in all the popular programming languages.

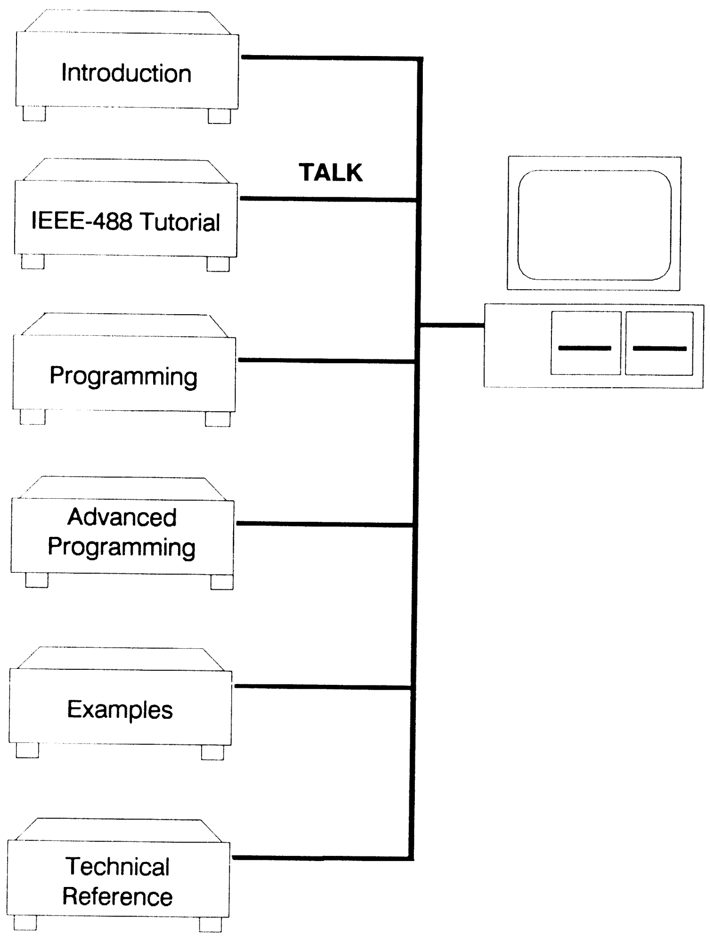
If you have no IEEE-488 experience, you may wish to read the tutorial in section 2. Topics in the tutorial build on each other and should be read in sequence. If you want a quick overview of the GPIB, there are only certain parts of the tutorial that are required reading. Topics that treat more advanced material are marked with an * in the table of contents and may be omitted the first time through.

CEC-488 can be programmed in two different ways: using the regular PRINT and INPUT statements of your programming language, or using subroutine CALLs directly to the on-board firmware. The first method is very simple to use, but limited in speed. The second method takes full advantage of the speed and power of the GPIB interface.

Here's how to choose the programming method you should use. IF your application involves:

- simple input and output of strings and variables, with speeds of around 1300 bytes/second being acceptable,
  - THEN use the PRINT and INPUT statements. See Appendix M for full details.
- advanced GPIB functions such as device triggering, multiple listening devices, high speed data transfer, detailed control over the GPIB (some instruments do not adhere fully to the IEEE-488 standard, and require more complex programming), as well as the simpler functions above,
  - THEN use the firmware CALLs. Section 3 describes the process of writing a program with CALLs to the CEC-488 firmware. Section 4 continues this discussion into more advanced topics. Once you have looked over section 3, you may wish to try the TRTEST program on the applications disk. TRTEST allows you to interactively send and receive data or commands and experiment with IEEE-488 devices.

Note: it is possible to mix the two methods of programming. You can start by using the simpler approach in Appendix M. If you find you need an advanced IEEE-488 function, such as serial polling, you can mix some firmware CALLs into your 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 (gp-ib) 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. 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.

An appropriate analogy for the organization of the gp-ib 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 gp-ib 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 gp-ib 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 gp-ib 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 gp-ib 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 gp-ib 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.

History

I want to thank everybody who made this day necessary - Yogi Berra

The gp-ib evolved out of a need for a common interface standard for engineering and scientific test equipment. In early 1972 there was an increasing demand for instrumentation systems that could be built from standard instruments. The advantage of sharing data between instruments was apparent but instruments were designed for stand-alone operation. This design deficiency resulted in several problems. Since instruments were designed for standalone operation, little documentation was available for systems design and the absence of a communications standard lead to prohibitive costs and delays in designing and configuring instrument systems. These problems coupled with the inflexibility of existing serial interface standards lead to "one-of-a-kind" systems that were generally unsupportable.

Given these problems, a set of objectives were formed out of a desperate need (and a healthy profit motive) for an effective system standard. These objectives included the following:

1. Electrical, mechanical, and functional interface requirements to allow systems to be built from independently manufactured devices.

2. Establish a set of common system terms and definitions.

3. Permit both simple (inexpensive) and complex instruments to work cost effectively in a single system.

4. Permit direct information exchange between instruments without the need for a central controller.

5. To allow asynchronous, limited distance communication with a minimum of restrictions on the performance of each device in the system.

6. To meet these objectives with a relatively low cost implementation that would be easy to use.

During March of 1972 a U.S. Advisory Committee (IEC) was formed to consider a proposal made by Hewlett-Packard Corporation that addressed these objectives. In September of 1974 the IEC approved a standard similiar to the one presented by Hewlett-Packard and it was published by the Institute for Electrical and Electronic Engineers (IEEE) in April of 1975. Since then there have been two revisions, one in 1978, and the other in 1980. Most of the changes were for clarification of the original standard.

The standard was quickly embraced by over 100 manufacturers who began to design instruments and systems using the IEEE document. The standard's popularity was based on its ability to to meet and exceed the original objectives. These objectives resulted in a long list of specifications. Some of the key specifications are shown below:

* Simultaneous data transfer from one source (talker) to multiple receivers (listeners).

* One megabyte per second data transfer rate (maximum) using 8 data lines and 8 bus management lines.

* Up to 15 devices on one system configured in linear or star topologies.

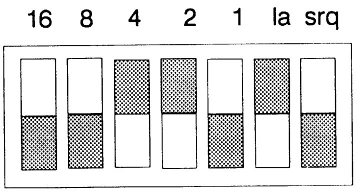
* Common interface functions within each device that require only passive cabling for system connections.

These specifications were also suitable for certain types of computer communications and the list of gp-ib compatible instruments soon included computer controllers and peripherals. Indeed, the rate of introduction of gp-ib compatible instruments, computers, and computer peripherals has increased every year since the acceptance of the standard. This growth has been stimulated by the introduction of very large scale integrated (VLSI) circuits that implement most of the system functions. The restriction of limited distance communication was eliminated with the introduction of gp-ib extenders that allow the bus to be extended to 1000 meters using coaxial cable or fiber optics. These extenders also allow communication over longer distances using telephone lines.

GP-IB Device Addresses

The gp-ib controller has the task of locating and communicating with all of the devices on the bus. To do this, each device on the bus must have an address by which it is identified. Once addresses are assigned, the controller can locate and define listeners, talkers, or other controllers.

Each address must be unique in order to allow individual devices to access the bus without interference. The bus address is usually assigned by external or internal switches, or by internal jumpers. Switch position, size, labeling, and definition are not covered by the IEEE standard, so they will vary from device to device. However, the address switch typically consists of five switches that allow the device to have one of thirty one addresses (0 through 30). The address location may also be set programmatically by the device microprocessor.



Typical address switch set for:
device address 6,
listen always,
service request disabled

Hints for setting device addresses

The IEEE 488-1978 standard does not allow a device address of 31. However, this address is used on some devices for diagnostic purposes.

It is often helpful to place a clearly visible address sticker on the front panel of the device. This is especially true if the device address has been changed from the factory default setting. Vendor supplied software may assume factory default addresses for certain devices and many frustrating hours of troubleshooting can be avoided by carefully setting the proper device address.

If a device address is changed while power is on, it is often necessary to turn power off and then on again before the address change will be recognized by the device.

* Secondary addressing

Secondary commands may be used to extend the total number of addresses on the bus. This does not increase the total number of devices that can be physically present, only the range of addresses that may be used. For example, only 31 primary addresses are allowed, so devices will have addresses 0 through 30 available to them. Sending a secondary address extends the addressing capability to 961 addresses (31 primary addresses in the first byte times 31 secondary addresses in the second byte).

Secondary addressing has a more subtle use than simply assigning an address outside of the 0 to 30 range. Secondary addresses are normally used to access additional operating modes on a single device. These extended operating modes are not defined by gp-ib protocol, so they must be determined from the device's operating manual. Any single device may have up to six secondary addresses. Secondary addresses are sent as ASCII characters ‘ through del (binary 01100000 through 01111111) with ATN asserted. Unlike primary addresses, secondary addresses do not distinguish between talkers and listeners. A device will recognize a secondary address if it is currently addressed to talk or listen (assumming that it is capable of interpreting a secondary address).

Refer to the ASCII table in the appendices for a complete cross reference of primary and secondary addresses and to the SEC command for techniques on programming with secondary addresses.

Listeners

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

Listeners are primarily data receivers. Printers, plotters, and programmable power supplies are all examples of listeners. Listeners may occasionally talk when they are asked for status information. For example, a host computer may want to know what type of printers are attached, whether or not they support graphics output, are online, out of paper, or unable to receive data because of an error condition. This type of information is usually transmitted from device to computer in response to a status request. Once addressed-to-listen, a device typically remains configured to listen until it receives an interface clear (IFC), its own talk address, or a universal unlisten (UNL) command.

A device can only be designated as a listener after it is assigned an address. The controller must determine or assume the device type and its address and then transmit an addressed-to-listen command to the device. Once this sequence occurs the device will be able to listen or "online".

Devices that are typically listeners may have a "listen always" or "listen only" switch next to the device address switches. A listen always mode is useful for detecting and logging any and all data message activity (control messages control the device and typically are not logged).

The listen only mode should be reserved for diagnostic purposes, since the switch overrides the unlisten command sent by a controller. Problems can arise when there are multiple listeners and one is set to listen always. For example, it is possible to check for bus activity by placing a printer in listen always mode. However, if a large file was transferred to a disk drive attached to the same bus, the printer would attempt to log the entire transaction. Since the data stream may contain commands that would change the character font, form feed, or printer mode, some exciting and unpredictable consequenses may result.

Up to fourteen devices can listen simultaneously even though the data may be accepted at different rates. If the transfer rate is unacceptably slow, the slow listeners can be bypassed by sending the universal unlisten command. On receipt of the command all devices will stop listening and the controller can designate specific (speedy) listeners.

The syntax and details of assigning listeners is discussed in section 3 - under LISTEN, TALK, and DATA commands.

Talkers

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

Talkers are primarily data transmitters. Digital voltmeters, analog to digital converters, and digitizers are all examples of talkers. Talkers must occasionally listen so that they will know when to begin and end transmission and when to change state. For example, a host computer may want to change the function, range, or trigger interval of the measurement device. Unlike listeners, there can only be one talker at a time.

A device can only be designated as a talker after it is assigned an address. The controller must determine or assume the device type and its address and then transmit an addressed-to-talk command to the device. When a device is addressed to talk it typically remains configured to talk until it receives an interface clear (IFC), another instrument's talk address, its own listen address, or a universal untalk command. While a new listener can be added to the bus without replacing a current listener, a new talker will always replace the current talker.

Talkers may have a "talk always" or "talk only" switch. This switch allows the talker to communicate with a listen only device without the need for a controller. If a talker is connected to a system with a controller, the talk only switch must be in the off position for the bus to work properly. A talker that could not listen would not have the capability to receive messages from the controller. Without this capability the talker would not know what type of information the listener or controller requires.

nterface Functions

Interface functions are the "glue" that holds the gp-ib together. Without them, no device could respond to a controller command, accept or receive data, or perform any other useful work on the gp-ib.

There are ten interface functions specified by the IEEE-488 standard. These interface functions specify how the device will operate in stand-alone or system environments. It is not necessary for a device to implement all of the functions, and most only implement a subset for reasons of economy. Therefore, most devices will respond to only a limited number of commands.

Some devices will list the functions that they implement next to the gp-ib connector. The function abbreviations that are most often used are shown below along with the function definition.

Source Handshake - SH this function provides a device with the capability to transfer messages from a talker to one or more listeners.

SH0 - no source handshake
SH1 - source handshake

Acceptor Handshake - AH this function provides a device with the ability to properly receive data from a talker.

AH0 - no acceptor handshake
AH1 - acceptor handshake

Talker or Extended Talker - Tx, TEx this function allows a device to send data and status messages when addressed to talk by the controller. A single-byte talk address establishes a talker, a two-byte address establishes an extended talker. Extended talker functions are shown in ().

T0 - not a talker or extended talker.
T1 - basic talker, serial poll, talk only mode.
T2 - basic talker, serial poll.
T3 - basic talker, talk only mode.
T4 - basic talker.
T5 - basic talker, serial poll, talk only mode, unaddress if MLA (unaddress if MSA and LPAS).
T6 - basic talker, serial poll, unaddress if MLA (unaddress if MSA and LPAS).
T7 - basic talker, talk only mode, unaddress if MLA (unaddress if MSA and LPAS).
T8 - basic talker, unaddress if MLA (unaddress if MSA and LPAS).

Listener or Extended Listener - L, LE this function allows a device to receive data and status messages when addressed to listen by the controller. A single byte talk address establishes a listener, a two byte address establishes an extended listener. Extended listener functions are shown in ().

L0, LE0 - not a listener or extended listener.
L1 - basic listener, listen only mode
L2 - basic listener
L3 - basic listener, listen only mode, unaddress if MTA (unaddress if MSA and TPAS).
L4 - basic listener, unaddress if MTA (unaddress if MSA and TPAS).

Service Request - SR this function allows a device to asynchronously request service from the controller. The service request will remain active until the device is serviced. If the service request consists of a multibyte transfer, service request will continue to be asserted until all bytes are read.

SR0 - no service request function
SR1 - service request function

Remote/Local - RL this function allows the device to select between two modes of operation - remote (via gp-ib) or local (front panel controls).

RL0 - no remote programming.
RL1 - complete remote-local programming functions.
RL2 - no local lockout.

Parallel Poll - PP this function allows a device to use one bit to identify its requirement for service in response to a parallel poll command from the controller. This function may not uniquely identify a device, since more than one device may be assigned the same bit. (Refer to Parallel Polling, in this section, for a more detailed explanation of this function.)

PP0 - no parallel poll function.
PP1 - remote parallel poll configuration.
PP2 - local parallel poll configuration.

Device Clear - DC this function allows a device to be cleared or initialized to a state defined by the device.

DC0 - no device clear.
DC1 - full device clear capability.
DC2 - omit selective device clear capability.

Device Trigger - DT this function allows a device to be triggered or synchronized with other devices.

DT0 - no device trigger.
DT1 - device trigger.

Controller - C this function allows a device to send bus commands and data and to address devices to talk or listen. It may also initiate serial or parallel polling. Any device on the bus can be a controller but there may only be one active controller at a time. Only one controller can be system controller (with the ability to take control of the bus by asserting IFC).

A controller is categorized according to its ability to perform the following functions:

C1 - System controller
C2 - Send IFC and take charge
C3 - Send REN
C4 - Respond to SRQ
C5 - Send interface messages
Receive control
Pass control
Pass control to self
Parallel Poll
Take control synchronously
C6 to 28 - various controller subsets based on combinations of the functions shown above.
C0 indicates that a device has no controller capability.

Another abbreviation on the connector declares the type of device bus driver. An E1 indicates an open collector driver and an E2 indicates a tri-state driver. E1/2 indicates a tri-state driver that can automatically switch to open collector drivers during a parallel poll.

CEC IEEE-488 interfaces have the following capabilities:

SH1, AH1, T5, TE5, L3, LE3, SR1, RL1, PP1, PP2, DC1, DT1, C1-5, E1/2.

The interpretation of remote versus local mode, the service request response, device clear, remote parallel poll configuration, and the device trigger function are software dependent.

Commands

The operation of the bus is usually controlled by one device designed to function as a controller. The controller transmits commands to direct other devices on the bus to carry out their functions as talkers and listeners. The controller has two ways of sending interface commands - multiline commands and single line commands. Multiline commands are sent over the eight data lines and use the three handshake lines DAV, NRFD, and NDAC to establish the command transfer timing. Single line commands are sent over five lines dedicated to managing the bus.

Commands serve two main purposes:

1. They select devices that will talk or listen. Addresses of the selected devices are always sent as multiline commands.

2. They force specific device actions. The actions are device dependent and are only generally defined by the IEEE-488 standard.

Although there are only two ways of sending interface commands, commands can be grouped into four categories,

1. Single line commands.
2. Multiline commands.
3. Addressed commands.
4. Secondary commands.

Single Line Commands

Single line commands are executed by toggling dedicated bus lines. Interface Clear (IFC) and Remote Enable (REN) are single line commands available only to the system controller. Attention (ATN) can be used by other controllers and End or Identify (EOI) is available to all devices.

Interface Clear (IFC)

All bus activity is unconditionally terminated when IFC is asserted. Only the system controller can assert IFC and the system controller becomes the active controller when IFC is asserted. All talkers and listeners become unaddressed and IFC overrides all other activity on the bus.

Remote Enable (REN)

Remote enable allows instruments on the bus to be programmed by the active controller (as opposed to being programmed only through the instrument controls). Any device that is addressed to listen while REN is true, may be programmed by the active controller.

Attention (ATN)

Attention enables a device on the bus to distinguish between commands and data.

If ATN is asserted then the data on the bus is a command. If ATN is not asserted then the data on the bus is a message. End or Identify (EOI)

End or identify is a special delimiter for the last data byte in a message (ATN must be false). Asserting EOI is not required by the IEEE-488 protocol and therefore some devices will use EOI while others will not. For the same reason, some controllers will assume that a data byte sent with EOI is the last byte of a message while others will not. Using EOI is recommended since a data byte itself (such as a carriage return or line feed) will not signal the end of a message to all devices. In many cases, it may be desireable to send a carriage return, line feed or other special character as part of the data message and EOI is the mechanism of choice for ending the message. The programming manuals for all devices in a system should be consulted to determine what delimiter is acceptable for all devices.

EOI and ATN are both true during an identify sequence in response to a parallel poll (refer to the topic - parallel polling in this section for further information). Multiline Commands (Universal Commands) Multiline commands cause specific messages to be placed on the data bus while ATN is asserted. These commands are called multiline, because they use more than one line (the data lines DIO1 to DIO8) to transfer the command, rather than a single dedicated bus-management line like IFC, REN, or EOI.

A controller executes a multiline command by asserting ATN while placing a specific message on the data bus. The results of the command and action taken by a device, is not specified in the IEEE standard. Every device has the option of taking no action at all, so it is necessary to check the user's manual to determine the results of any multiline command. The definitions given below are only approximate, since the specific action taken by a device is determined by the manufacturer and not by the IEEE standard.

These commands are commonly referred to as universal commands, since they go out to all devices on the gp-ib. They comprise the universal command set as defined by the IEEE standard.

Device Clear (DCL)

On receipt of this command the device is initialized to its power-on state. The specific state of the device is defined by the manufacturer and not by the IEEE-488 standard.

Local Lockout (LLO)

When the local lockout message is sent the device cannot be programmed from its front panel. If front panel control is required the controller can send a go-to-local (GTL) command. It is also good programming practice to send a (GTL) command after the controller has completed remote programming in the local lockout state. Automated systems can maintain data integrity by using local lockout at appropriate times to prevent inappropriate or untimely setting of front panel controls.

Parallel Poll Unconfigure (PPU)

This command resets all parallel poll devices to the idle state. A device will not respond to a parallel poll after receiving this command.

Serial Poll Enable (SPE)

On receipt of this command the device will assert the service request line when it requires service and provide one or more data bytes to indicate its status when polled by the controller. Serial Poll Disable (SPD)

The serial poll disable command terminates serial poll mode for all responding devices. SPD returns the devices to their normal talker state.

Addressed Commands

Addressed commands (also referred to as primary commands) are similiar to universal commands, except that only devices that have been addressed to listen will respond to the command. Addressed commands are always multiline commands. A controller executes an addressed command by asserting ATN while placing a specific message on the data bus.

The results of an addressed command and the action taken by a device, is not specified in the IEEE standard. Every device has the option of taking no action at all, so it is necessary to check the user's manual to determine the results of any addressed command. For this reason, the definitions given below are only approximate.

Group Execute Trigger (GET)

GET is used to simultaneously trigger a group of devices. Response to a trigger message is completely device dependent. The controller cannot indicate in the message what action should be taken.

Selected Device Clear (SDC)

On receipt of this command a device currently addressed to listen is intialized to its power-on state. The difference between SDC and DCL is that DCL clears all clearable devices, while SDC clears only the active listeners.

Go To Local (GTL)

On receipt of this command a device may resume control from its front panel.

Parallel Poll Configure (PPC)

This command allows the controller to remotely program the parallel poll response of a specific device. Some devices will respond to a parallel poll request even though they will not respond to a parallel poll configure. Precautions should be taken in using the PPC command to assure a unique response from all devices.

Take Control (TCT)

This command allows the current controller to pass control to a new controller.

Secondary Commands

Parallel Poll Enable (PPE)

A parallel poll enable is only meaningful if each device on the bus has been configured for a specific parallel poll response. To configure the bus, the controller must assert attention, establish itself as a talker, and tell all other devices to unlisten. At that point it can address individual devices (addressed as listeners), provide them with their parallel poll response, and issue the parallel poll enable command.

Parallel Poll Disable (PPD)

The parallel poll disable command is targeted at specific devices by first addressing the device and then issuing the parallel poll disable command.

Information on the bus is interpreted as data whenever attention (ATN) is not asserted.

There are no restrictions on what data characters may be sent and there are no limits to the length of data messages. However, sending data continuously may result in poor overall performance for the controller if there is one slow listener on the bus. A typical example would be a computer driving a slow wordprocessing printer. If the computer is capable of running multiple tasks, overall performance will improve if the computer runs other tasks until the printer interrupts with a request for more data.

This type of optimization is not required in BASIC programming since the data will typically be accepted by all listeners before another data-transfer statement can be interpreted by the microprocessor.

Bus Traffic Types - a summary

The bus traffic types discussed so far are:

Bus commands
Listen addresses
Talk addresses
Secondary addresses
Data messages

Bus commands are a set of ASCII characters transferred with the attention line (ATN) asserted. These commands consist of the multiline or universal commands (DCL, LLO, PPU, SPE, SPD), the addressed commands (GET, SDC, GTL, PPC, TCT), and the secondary commands (PPE, PPD). Instruments or peripherals may respond to all, some, or none of these commands.

Listen addresses are a set of ASCII characters transferred with ATN asserted. A device will receive data after recognizing its listen address.

Talk addresses are a set of ASCII characters transferred with ATN asserted. A device will send data after recognizing its talk address. A device will stop sending data when it recognizes a talk address different from its own.

Secondary addresses are a set of ASCII characters transferred with ATN asserted. Secondary addresses are normally used to access additional operating modes on a single device.

Data messages are sent with ATN unasserted. Data is transferred from a talker to a listener or listeners at a rate determined by the slowest listener. Every device listens to bus commands, but only devices addressed to listen receive data. A special type of data message is the terminator or delimiter. This message is sent at the end of a data block with EOI asserted.

System Controller

Leadership is action, not position. -Donald McGannon

The PC is usually configured as a system controller. The system controller is the active controller or moderator for the system and has responsibility for assigning a talker and listeners, issuing bus management commands, and directing the overall flow of data on the bus.

The system controller is the only device that can assert interface clear (IFC) and remote enable (REN). All other single line, multiline, addressed, and secondary commands may be issued by any active controller.

Asserting IFC usually gives the system controller unconditional control of the bus. However, there is one circumstance where the system controller will not gain immediate control. If ATN is asserted by another controller, the interface will wait until ATN is no longer asserted and then become the active controller.

Passing Control (The computer as a device)

If there is more than one device on the gp-ib with controller capability, then the current active controller can pass control to another controller. The ability to pass control allows computers to independently control shared I/O devices such as printers, disk drives, or instruments. Local area networking, with multiple controllers, is a powerful feature of the gp-ib.

When the computer passes control it takes on different responsibilities and has fewer privileges than it had as a controller.

It must - Determine what functions it will implement.

Monitor its listen and talk status and take appropriate action when addressed to listen or talk.

Respond to a take control command if it is to regain control of the bus without asserting IFC. It cannot - Perform any bus addressing or send commands.

Passing control from the current active controller to a new active controller is a three step process.

1. The active controller must address the prospective active controller to talk.

2. The current active controller sends the take control (TCT) message.

3. If the prospective controller accepts the message, then it becomes the new active controller and the original controller becomes a non-active controller.

Conceptually, this process is straightforward, however, these three steps make several assumptions about the state of the bus and the state of the prospective active controller. For example, it assumes that the prospective active controller has the ability to become active controller. It also assumes that when a controller becomes a device it can talk or listen when addressed to talk or listen. That it can respond appropriately to bus commands and only take control of the bus when it is asked to take control. These assumptions and the process of passing control are more thoroughly discussed in section 4 under the topics - The Computer as a Device and Passing Control.

The system controller always retains the ability to take unconditional control of the bus by asserting interface clear (IFC).

* Serial Polling

The other line moves faster. -Barbara Ettore

Serial polling is a method of sequentially checking status on any number of devices on the gp-ib. To do this the service request line (SRQ) controlled by the device and the serial poll, performed by the controller, must work together. The controller enables a serial poll by addressing a device to listen, followed by a device dependent command that enables (or unmasks) various conditions that will cause the device to assert SRQ. After a device is enabled, it can assert the service request line anytime it needs service. This is the only method a device can use to initiate an action by the controller.

If there is only one device on the gp-ib enabled to assert SRQ, the controller can read the device's status register and perform any required service. If there is more than one device on the gp-ib enabled to assert SRQ, then the controller must sequentially read the status register of each device. This is because the service request line is shared (wire ORed) between all devices and the controller cannot determine the source of the request from the service request line alone.

For example, assume there are several devices on the bus and a serial poll is to be taken on all devices. When the service request line (SRQ) is asserted, the controller begins to sequentially gather status from each device. One feature of the serial poll is that the status information returned to the controller contains the service request status and information about the event that caused the service request. This single status byte is usually sufficient to determine the requirements of the device.

The primary feature of the serial poll is that it allows the controller to assign tasks to several devices without being required to wait for a device response. It also eliminates the need for the computer to constantly check the interface status. This can significantly improve controller efficiency in systems with slow devices.

When a serial poll is performed, the resulting bus activity is shown below:

- Unlisten (UNL command)
- My listen address (MLA computer)
- device's talk address (one of the talk address group commands)
- serial poll enable (SPE)
- one data byte is read by the computer (the status byte)
- serial poll disable (SPD)
- untalk (UNT command)

Note that a serial poll enable and serial poll disable have no effect on a device's ability to assert SRQ. The ability to assert SRQ is determined by a device dependent instruction sent to the device by the controller.

Some devices provide a serial poll mask. The mask is set by sending a command to the device that allows the controller to define what conditions within the device will produce a service request. If the device has a mask, the mask default mode and programming method should be understood before attempting a serial poll. It is quite likely that the device initially masks all service request conditions; leaving initialization to the controller (you).

If a device is enabled for a serial poll it will assert the service request line (SRQ) until it is serviced. After a device has been serviced, the IEEE standard requires that the device stop requesting service until a new condition arises (or the same condition arises again). However, serial polling (or servicing) may require reading more than one status byte. If a device continues to assert SRQ after the first status byte has been read, it is because additional information is still available in the device's status register. The number, order, and content of all status bytes must be known by the controller to perform a successful serial poll. SRQ may also remain asserted after all status bytes have been read from a device, if subsequent devices request service during the serial poll procedure.

* Serial polling and the Service Request Interrupt

There are times when the controller can be made more efficient by performing multiple tasks simultaneously. The gp-ib has built-in provisions that will allow the controller to program certain device dependent actions and then begin performing another task without waiting for the device to complete its action or return data. This can be done if the device implements the service request interrupt and the controller has enabled a serial poll or enabled and configured a parallel poll.

For example, the controller might want to print a long file that would overrun the buffer memory of a slow printer. If a serial or parallel poll is enabled the controller can send data until the printer can no longer accept data and then the controller can begin a new task. When the printer buffer is empty the printer will assert the service request line (SRQ) and interrupt the controller from its current task. The controller then begins a serial or parallel poll to determine what device needs service. If the controller is performing a serial poll the printer would respond with a status byte. Information in the status byte will indicate to the controller why the printer requires service. If the controller is performing a parallel poll, the printer would assert a single bit on the bus that the controller recognizes as the printer request bit.

* Parallel Polling

Parallel polling is a method of simultaneously checking status on up to eight instruments on the bus. When a parallel poll is initiated, each device returns a status bit via an assigned data line. Both the sense (high or low) and the data line are assigned by the controller with a parallel poll configuration routine. (See the programming examples section for details).

If the status bits returned during a parallel poll indicate that no device requires service, then the bus is in the parallel poll idle state. It is the responsibility of the controller to periodically perform a parallel poll to determine if any devices need service.

Serial and parallel polling are often done in sequence in order to take advantage of the strengths of both techniques. Parallel polling quickly determines the device that requires service and serial polling quickly determines the type of service required. A controller will typically enable the service request interrupt of all devices on the gp-ib and then process the interrupt with sequential parallel and serial polls.

There are some cases where a sequential parallel and serial poll is required. This occurs when more than one device is assigned to respond on a single bit.

For example, assume that there are ten devices on the bus and a parallel poll is to be taken on all devices. Six devices would be assigned a unique bit and two pairs of devices would have to share two bits. If one of the shared bits is asserted during a parallel poll then the computer must serially request the status of the two devices sharing the bit.

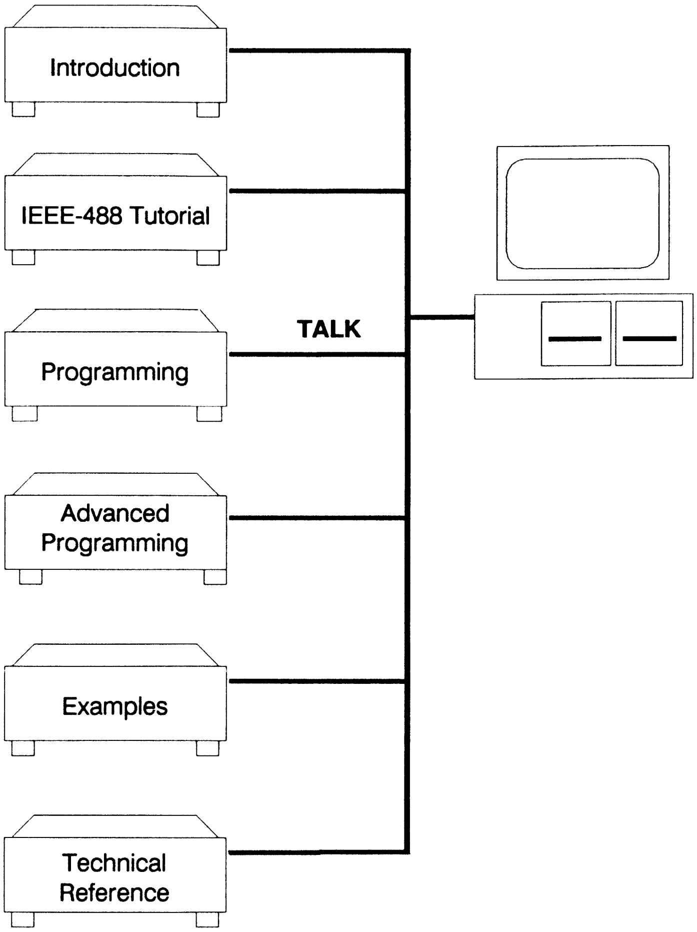
Assignment of the shared bits should go to devices that require the least frequent service. However, some devices may automatically assign their parallel poll bit to correspond to their primary address (or an arbitrary address choosen by the manufacturer). This allows the manufacturer to simplify both hardware and software but it limits the controller's ability to configure the system. If a device automatically assigns its parallel poll bit it is likely that a primary address greater than seven will result in no parallel poll response. This places another restriction on systems with more than eight devices and the owner's manual of each device should be consulted for information concerning restrictions or limitations.

Some devices provide a parallel poll mask. The mask is set by sending a command to the device that allows the controller to define what conditions within the device will produce a service request. If the device has a mask, the mask default mode and programming method should be understood before attempting a parallel poll. It is quite likely that the default mode turns off any parallel poll response, leaving initialization up to the controller.

In some systems the controller will perform a continuous parallel poll. It is perfectly acceptable for no device to respond and this corresponds to the parallel-poll-idle state of the IEEE-488 standard.

Precautions - if there is more than one device on the bus and one is in talk only mode, a parallel poll request could result in devices trying to drive the bus in opposite directions. The device that is in talk only mode may not respond to the parallel poll request and continue talking. If other devices do respond to the request, the resulting bus conflict would produce ambiguous data.

There is no guarantee that consecutive parallel polls will return the same information, even if the controller takes no action to service a device. This can happen because it is possible for a device to update its parallel-poll hardware registers asynchronously and independently of any action taken by the controller.



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 CEC-488 for instrument control applications.

Examples in this section and the following section are given in the programming languages most popular with our customers: BASIC, QuickBASIC, Turbo Pascal, and C. CEC-488 can be programmed in other languages using the same steps presented in this section. The appendices 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 appendices. Insert the CEC-488 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 CEC-488 commands, for more sophisticated GPIB control, are introduced later.

The software described in this manual works with 4x488, PS< >488, and PC< >488. CEC also manufactured an older version of PC< >488 based on a different hardware design, which will not work with this software (except the BASICA interface, which will work). The older design can be identified by its lack of the two sets of DIP switches present on the newer PC< >488.

Programming CEC-488 in BASICA or GWBASIC

CEC-488 uses a read-only-memory (ROM) that contains GPIB language extensions for BASIC. BASIC uses the CALL statement to access those language extensions. To use the CALL statement, you must first provide a "DEF SEG" statement (as shown in line 10 on page 3-5) which defines the memory segment address of the CEC-488 ROM. The memory address is a hexadecimal value so it is preceeded by "&H". The memory address is set by CEC to a position that is compatible with the widest range of PC configurations. If you have a memory conflict the address can be changed (see Hardware Configuration appendix).

Note: if you have a PS< >488 card, you have a RAM instead of ROM, and you must load the software before running any BASIC program. To load the software, run the program PS488 on the application disk. You can copy this program to your system disk, and add it to your DOS AUTOEXEC.BAT file if you like.

In order to call specific routines within the ROM, BASIC needs to know the address of the routine. This is called an offset address since it is the number of bytes the routine is offset from the DEF SEG address. The offset for the INITIALIZE routine (line 20, page 3-5) is zero. One segment address is required in every program and an offset address is required for each CEC-488 routine that you use.

Each CEC-488 routine has a specific list of arguments (or variables) that it needs. These arguments are always integers or strings and they are enclosed in parentheses in the CALL statement. BASIC variable names ending with a percent sign (%) are integers. BASIC variables ending with a dollar sign ($) are strings. BASIC also provides the "DEFINT" statement to allow you to define an entire range of variable names to be integer variables. For example,

```
5 DEFINT A-H
```

will define all variable names beginning with letters from A through H as integers. Variables must be assigned the correct type or your program will not run properly.

Initializing the GPIB

The first step in any GPIB control program is to initialize the system. This is done with the CEC-488 **Initialize** routine.

The **Initialize** routine is called with two arguments: the first gives the GPIB address you wish to assign to the CEC-488, and the second specifies whether CEC-488 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 CEC-488. 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 CEC-488 interface for operation. Second, it sets the GPIB address to be used by CEC-488. 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 PC< >488, switch S1 position 8 is set by CEC to the off position to make PC< >488 a system controller. This allows PC< >488 to send GPIB bus commands. If you want PC< >488 to be a device (receive commands only) S1 position 8 should be ON.

BASICA or GWBASIC:

```
10 DEF SEG=&HCC00          ' memory address
20 INITIALIZE=0            ' subroutine offset
30 my.address%=21          ' CEC-488 GPIB address
40 controller%=0           ' system control
50 CALL INITIALIZE (my.address%,controller%)
```

QuickBASIC:

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

Turbo Pascal:

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

C:

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

Sending Data to a Device

Once the GPIB system has been initialized, the next step is usually to send a command or data to a device. The CEC-488 **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).
- 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 CEC-488 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.

BASICA or GWBASIC:

```
55 SEND=9              ' offset for SEND
60 s$="MEASURE"        ' command for device
70 address%=7          ' GPIB address of the device
80 CALL SEND (address%,s$,status%)
```

QuickBASIC:

```
CALL SEND (7,"MEASURE",status%)
```

Turbo Pascal:

```
send (7,'MEASURE',status);
```

C:

```
send (7,"MEASURE",&status);
```

Reading Data from a Device

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 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.
- address is the GPIB address of the device to read from.
- status indicates whether the transfer went OK.
  0 = okay
  8 = timeout

Before **Enter** is called, an area must be reserved for the received data. In BASIC, the best way to do this is to set a string variable equal to an appropriate number of spaces with the SPACE$ function. SPACE$ takes one argument giving the number of spaces. This should be at least equal to the maximum length of data expected. In C or Pascal, simply declare the string variable normally.

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

BASICA or GWBASIC:

```
90 ENTER=21                 ' offset for ENTER
100 r$=SPACE$(30)           ' allocate room for data
110 address%=7              ' GPIB address of the device
120 CALL ENTER (r$,length%,address%,status%)
130 r$=LEFT$(r$,length%)     ' trim string to length
```

(Because BASIC does not allow called routines to actually modify the length of a string argument, you should trim the received string to the length returned by **Enter** (line 130)).

QuickBASIC:

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

Turbo Pascal:

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

C:

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

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

BASICA or GWBASIC:

```
5  '-- Example instrument control program
6  '
10 DEF SEG=&HCC00             ' memory address
20 INITIALIZE=0 : SEND=9 : ENTER=21 ' subroutine offsets
25 '
26 '-- Initialize the GPIB system
27 '
30 my.address%=21             ' CEC-488 GPIB address
40 controller%=0             ' system control
50 CALL INITIALIZE (my.address%,controller%)
55 '
56 '-- Send a command  (take a measurement)
57 '
60 s$="MEASURE"              ' command for device
70 address%=7               ' GPIB address of the device
80 CALL SEND (address%,s$,status%)
82 IF status%<>0 THEN PRINT status% : STOP
85 '
86 '-- Read the measured value
87 '
90  r$=SPACE$(30)             ' allocate room for data
100 CALL ENTER (r$,length%,address%,status%)
102 IF status%<>0 THEN PRINT status% : STOP
110 r$=LEFT$(r$,length%)       ' trim string to length
115 '
120 PRINT "Value is ";VAL(r$)
```

If the data string received from a device represents a number (for example, the string " 12.45"), it can be converted to a numeric variable with the VAL function in BASIC (see line 120).

If a received data string contains multiple numbers separated by a known character, such as a comma or semi-colon, you can define the FNCHOP$ function as shown below to extract the separate values:

```
130 DEF FNCHOP$(X$,Y$) = RIGHT$(X$,LEN(X$)-INSTR(X$,Y$))
140 r$=SPACE$(30)
150 CALL ENTER (r$,length%,address%,status%)
160 X = VAL(r$)                ' first value
170 Y = VAL(FNCHOP$(r$,",",))          ' second value
180 Z = VAL(FNCHOP$(FNCHOP$(r$,","),",")) ' third value
```

If, in this program, R$ contains " 12,-34.5, 78", X will contain the value 12, Y will contain -34.5, and Z will contain 78. The FNCHOP$ function returns its first string argument with the portion up to the first occurence of the second string argument removed. Thus, FNCHOP$(R$,",") is "-34.5, 78".

QuickBASIC:

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

Turbo Pascal:

```
PROGRAM example;
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:

```
#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);
}
```

Checking the Device Status - Service Request

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).

Note: BASICA does not support the **Srq** function. You can test the GPIB hardware interface chip directly for the SRQ status bit by doing an input operation from port 2BA.

```
300 IF (INP(&H2BA) AND &H40)=0 THEN 300
```

QuickBASIC:

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

Turbo Pascal:

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

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 on the next page, the program tests for service request and stays in a loop calling **Spoll** until the status has one particular bit true.

BASICA or GWBASIC:

```
82 SPOLL=12          ' offset for SPOLL
84 ADDRESS%=7        ' GPIB address of the device
86 CALL SPOLL (ADDRESS%,POLL%,STATUS%)
88 IF (POLL% AND 4)=0 THEN 86
```

QuickBASIC:

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

Turbo Pascal:

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

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).

BASICA or GWBASIC:

```
140 PPOLL=15       ' offset for PPOLL
150 CALL PPOLL(poll%)
160 PRINT "parallel poll is ";poll%
```

QuickBASIC:

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

Turbo Pascal:

```
ppoll (poll);
```

C:

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

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,

- CEC-488 must be the system controller.
  - **Send** and **Enter** assume that CEC-488 is the GPIB controller. If there is another controller and CEC-488 is being used as a device you should use **Transmit** and **Receive.**
- 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.
  - CEC-488 provides **Tarray, Rarray**, and DMA2 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.

Low-level GPIB Control using Transmit

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.

BASICA or GWBASIC:

```
170 TRANSMIT=3                   ' offset for TRANSMIT
180 t$="UNL LISTEN 4 7 GET" ' command string
190 CALL TRANSMIT (t$,status%)
```

QuickBASIC:

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

Turbo Pascal:

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

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 the CEC-488 as the talker.

MLA

My Listen Address. Assigns the CEC-488 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.

Transmit Examples

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

```
195 VOLTAGE=3.5
200 CMD$="MTA LISTEN 1 DATA 'V="+STR$(VOLTAGE)+"' 13 10"
210 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 line 200 executes, CMD$ will be "MTA LISTEN 1 DATA 'V= 3.5' 13 10".

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.

Serial poll setup

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.

Parallel poll setup

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.

Other multiline commands

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).

Other addressed commands

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 Receive Routine

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 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.

BASICA or GWBASIC:

```
200 cmd$="MLA TALK 8"     ' set device 8 to talk
210 CALL TRANSMIT (cmd$,status%)
220 RECEIVE=6             ' offset for RECEIVE
230 r$=SPACE$(30)         ' make room for data
240 CALL RECEIVE (r$,length%,status%)
250 r$=LEFT$(r$,length%)
```

QuickBASIC:

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

Turbo Pascal:

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

C:

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

Binary Data Transfers

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 up to 64K bytes 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. In some languages like BASIC, the memory address of the data array must be passed to **Tarray**. Often this memory address consists of two components, a segment and an offset. See the examples and the language interface appendices for more information.
- 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.

BASICA or GWBASIC:

```
290 DIM INFO%(1000)        ' array containing data
... fill the array with data ...
300 cmd$="MTA LISTEN 8"        ' set up listener
310 CALL TRANSMIT (cmd$,status%)
315 TARRAY=200             ' offset for TARRAY
320 count%=2000            ' send 2000 bytes of data
330 eoi%=1                 ' send EOI with last byte
340 seg%=-1 : ofs%=VARPTR(INFO%(1)) ' get address
350 CALL TARRAY (seg%,ofs%,count%,eoi%,status%)
```

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.

QuickBASIC (version 4.0 or later):

```
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:

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

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. In some languages like BASIC, the memory address of the data array must be passed to **Rarray**. Often this memory address consists of two components, a segment and an offset. See the examples and the language interface appendices for more information.
- 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.

BASICA or GWBASIC:

```
290 DIM INFO%(1000)          ' array containing data
300 cmd$="MLA TALK 8"        ' set up talker
310 CALL TRANSMIT (cmd$,status%)
315 RARRAY=203               ' offset for RARRAY
320 count%=2000              ' send 2000 bytes of data
330 eoi%=1                   ' send EOI with last byte
340 seg%=-1 : ofs%=VARPTR(INFO%(1)) ' get address
350 CALL RARRAY (seg%,ofs%,count%,length%,status%)
```

QuickBASIC (version 4.0 or later):

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

Turbo Pascal:

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

C:

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

In most cases, you should declare the data array 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 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. CEC's companion product **CEC-Graph™** can be helpful here, since it includes the CONVERT procedure, which automates such conversions.

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.

High Speed Data Transfers

CEC's IEEE-488 boards can use the direct memory access (DMA) channels in the computer to transfer data at much higher speeds than is possible under the control of the PC's microprocessor.

PC's and PC/XT's have 4 DMA channels, numbered 0 to 3. PC/AT's and the Micro Channel PS/2's have 8 channels, numbered 0 to 7. Some of these channels are used by hardware built in to the computer. For example, channel 0 in PC's, XT's, and AT's is used to maintain (refresh) system memory. CEC-488 boards usually come with DMA channel one configured, but the channel may be changed (see Hardware Configuration appendix).

DMA may be used with the **Tarray** and **Rarray** routines to speed up long data transfers. DMA is initially disabled, and must be enabled with the **Dma-Channel** 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.

Transfer speed is usually limited by either the speed of the GPIB device or by the DMA hardware inside the computer. Most PCs acheive a DMA rate of around 300-400K bytes/second. Since most GPIB instruments go slower than this rate, no special measures are necessary. DMA speed can be improved by using "compressed mode" DMA timing, which eliminates some wait states during DMA transfers. Compressed mode is set by outputting a value of 8 to I/O port 8 before starting the DMA transfer. It is turned off by outputting a zero to port 8.

Compressed mode timing does not work correctly with the system RAM in all computers, particularly the older PC, XT, and AT designs. It is guaranteed to work correctly with the optional cache RAM on PC< >488. If you are using the PC< >488 board and have the cache RAM installed, you can specify its memory address as the location for the transfer and use compressed mode. You can then move the data from the cache RAM to regular system RAM with the computer's memory move instruction, which runs faster than DMA rates.

BASICA or GWBASIC:

DmaChannel is not available in this language. See the BASIC language appendix for information on how to do DMA transfers in BASIC using the DMA2 routine.

QuickBASIC:

```
CALL DMACHANNEL(1)
```

Turbo Pascal:

```
dmachannel(1);
```

C:

```
dmachannel(1);
```

Configuring Board Parameters

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 CEC-488 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 CEC-488 interfaces are used.

BASICA or GWBASIC:

SetPort is not available in this language.

QuickBASIC:

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

Turbo Pascal:

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

C:

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

BOARDSELECT (board)

where:

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

BoardSelect is used only when multiple CEC-488 interfaces are installed in the computer.

BASICA or GWBASIC:

BoardSelect is not available in this language.

QuickBASIC:

```
CALL BOARDSELECT (2)
```

Turbo Pascal:

```
boardselect (2);
```

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.

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.

BASICA or GWBASIC:

SetTimeout is not available in this language.

QuickBASIC:

```
CALL SETTIMEOUT (3000)
```

Turbo Pascal:

```
settimeout (3000);
```

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.

BASICA and GWBASIC:

SetOutputEOS is not available in this language.

QuickBASIC:

```
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.

BASICA and GWBASIC:

SetInputEOS is not available in this language.

QuickBASIC:

```
CALL SETINPUTEOS (13)
```

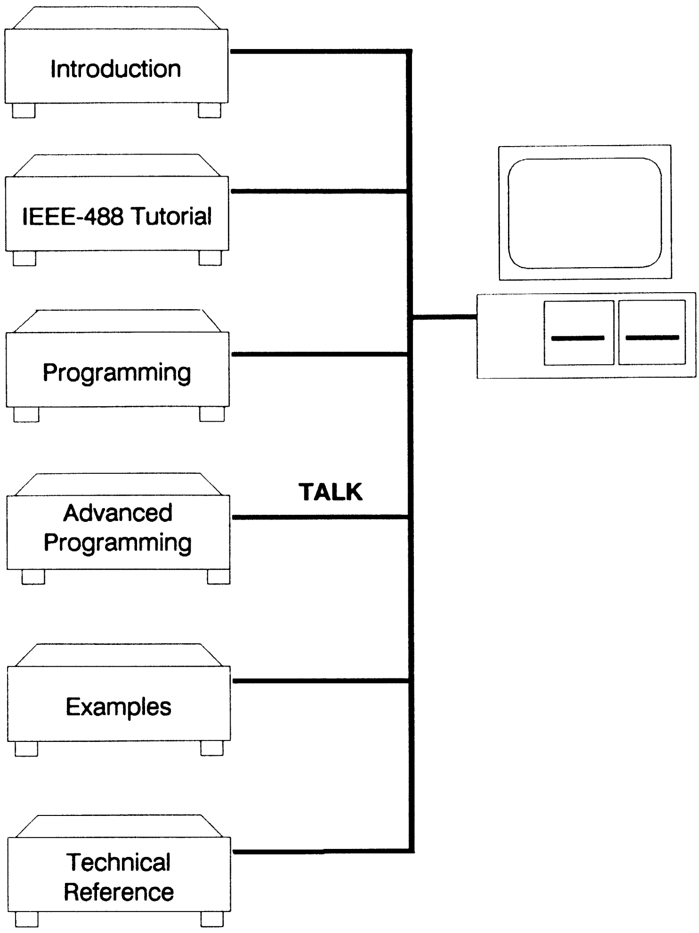
Turbo Pascal:

```
setinputeos (13);
```

C:

```
setinputeos (13);
```

Advanced Programming



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 as a GPIB Device

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 CEC-488's GPIB address is handled through the **Initialize** routine. When **Initialize** is called with its second argument equal to two, CEC-488 will act as a device. Note that PC< >488 also needs to have switch S1 position 8 turned ON to become a device. For 4x488, you should run the CECCTRL program provided on the Installation disk, with the command "CECCTRL OFF". PS< >488 will automatically switch to device mode when you call **Initialize.**

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:

BASICA and GWBASIC:

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

QuickBASIC:

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

Turbo Pascal:

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

Microsoft C:

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

Input:

Address (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 CEC-488 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.

BASICA, GWBASIC, or QuickBASIC:

```
90 '--- wait until addressed to talk or listen
100 IF (INP(&H2BC) AND 2)<>0 THEN 1000
110 IF (INP(&H2BC) AND 4)<>0 THEN 2000
120 GOTO 100
1000 '--- TALK
...
2000 '--- 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;
```

Microsoft 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 ---
7 '
10 DEF SEG=&HCC00
20 INIT=0 : TRANSMIT=3 : RECEIVE=6    ' offsets
30 MY.ADDRESS%=2 : DEVICE%=2
40 CALL INIT (MY.ADDRESS%,DEVICE%)    ' initialize
45 '
47 COMMAND=0
48 '
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
90  '
1000 '--- TALK ---
1002 '
1005 IF COMMAND=0 THEN RETURN    ' not yet told to
measure
1010 T$="DATA 'data from the PC' 13 END'"
1020 CALL TRANSMIT (T$,STATUS%)
1025 COMMAND=0
1030 RETURN
1040 '
2000 '--- LISTEN ---
2005 '
2007 IF COMMAND=1 THEN RETURN    ' measurement already
read
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=&HCC00
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 sub-routine 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 sub-routine 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$="" THEN EXIT SUB  ' nothing to say?
  IF (STATUS AND 2)=0 THEN EXIT SUB ' not ready to send?
  STATUS=STATUS AND &HFD      ' clear status bit
  OUT &H2B8,ASC(OUTBUF$)      ' 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 in buffer?
  CALL GetStatus        ' get status
  IF (STATUS AND 1)=0 THEN EXIT SUB ' no byte received?
  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.
```

Microsoft 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.

```
100 OUT &H2BB,&H41    ' set spoll response and
102                ' 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.

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
```

Passing Control

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 the CEC-488 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 the CEC-488. A description of the interrupt conditions available is given under "The Computer as a GPIB Device", earlier in this manual.

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

- The CEC-488 must have the appropriate hardware interrupt jumper installed. Interrupts 2 through 7 may be chosen (see the Hardware Configuration appendix of the manual for the jumper locations). Note that other hardware on the PC may use interrupts, and that the CEC-488 must not conflict with these. The CEC-488 comes with interrupts disabled. Interrupt 3 should be OK if you don't have two serial (COM:) ports.
- The CEC-488 must be initialized, using the INITIALIZE firmware 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 the CEC-488 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:

```
;... 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 BASIC, Capital Equipment Corp. has a software product available (order number 01000-10300, BASIC Interrupt Software).

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 CEC-488 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.
```

Using Multiple Boards

Multiple CEC-488 interfaces can be used in the same computer. This can be useful when more than 14 devices must be controlled.

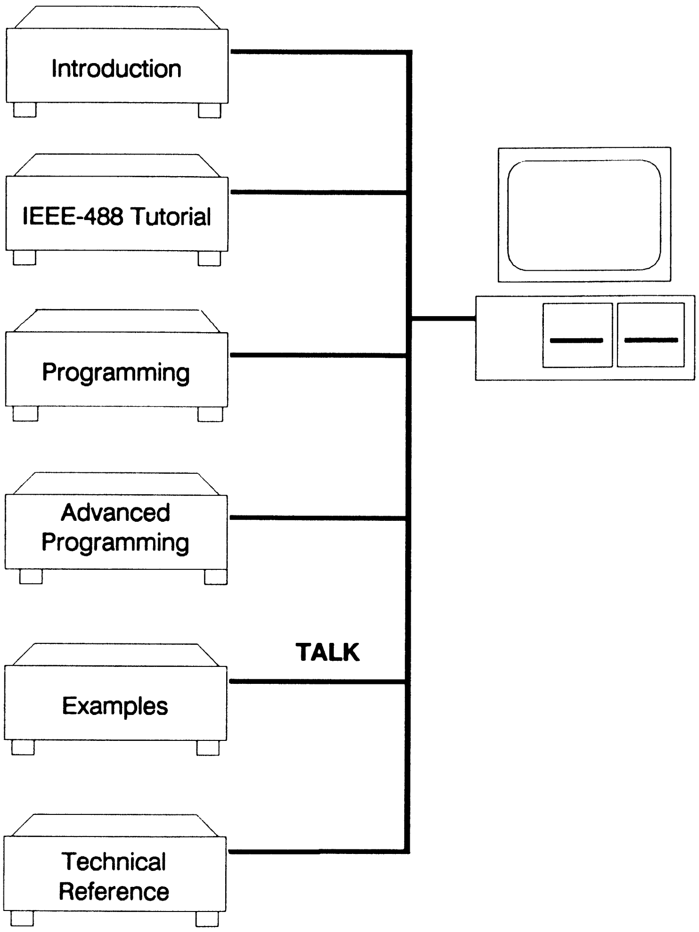
When more than one CEC-488 is installed in a computer, they must not use the same memory address, I/O address, interrupt channel, or DMA channel. See the Hardware Configuration appendix for details on setting these addresses and channels.

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%)
```

Programming Examples



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 CEC-488. 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 applica- tions 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  '--------------------------------------------------
50  DEF SEG=&HCC00              ' memory segment for board
60  INITIALIZE=0                ' offsets for subroutines
70  SEND=9 : ENTER=21
80  '
90  MY.ADDRESS%=21  ' make PC a controller at address 21
100 LEVEL%=0
110 CALL INITIALIZE(MY.ADDRESS%,LEVEL%)
120 '
130 ADDRESS%=16         ' GPIB address of the instrument
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 '----------------------------------
40 DEF SEG=&HCC00
50 INITIALIZE=0 : SEND=9 : ENTER=21
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 ---
116 '   (see "The computer as a device" in the manual
117 '     for a list of all the status bits)
124 '
125 IF (INP(&H2BA) AND &H40)=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 '----------------------
40 DEF SEG=&HCC00
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  '----------------------------------------------
40  DEF SEG=&HCC00
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 at addresses
100 ' 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 '---------------------------------------------
50  DEF SEG=&HCC00
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
```

```
10 '************************************************
20 '       Tektronix 5010 series demo
23 '    Uses a Tek DC5010, DM5010, and FG5010
24 '    to generate waveforms and measure the frequency
25 '    and amplitude.
30 '************************************************
40 CLS : KEY OFF
60 PRINT
70 PRINT "Voltage","Frequency"
90 ' Init interface hardware & software
110 DEF SEG=&HCC00
120 INIT=0 : ADDR%=21 : LEVEL%=0
130 CALL INIT(ADDR%,LEVEL%)
150 ' Initialize devices
170 SEND=9 : ENTER=21
180 FG5010%=24 : DM5010%=16 : DC5010%=20    ' addresses
190 S$="INIT;OUT ON" : CALL SEND(FG5010%,S$,STATUS%)
200 S$="INIT;ACV"    : CALL SEND(DM5010%,S$,STATUS%)
210 S$="INIT"        : CALL SEND(DC5010%,S$,STATUS%)
230 ' Measurement loop
250 DC5010$="FREQ;SEND"
260 DM5010$="ACV;SEND"
270 RD$=SPACE$(255)
280 AMPLITUDE = 1
290 FOR FREQUENCY = 1000 TO 2500 STEP 100
300   S$="FREQ "+STR$(FREQUENCY)+";AMP "+STR$(AMPLITUDE)
310   CALL SEND (FG5010%,S$,STATUS%)
320   CALL SEND (DC5010%,DC5010$,STATUS%)
330   CALL SEND (DM5010%,DM5010$,STATUS%)
340   CALL ENTER (RD$,LENGTH%,DM5010%,STATUS%)
350   DM=VAL(LEFT$(RD$,LENGTH%))
360   CALL ENTER (RD$,LENGTH%,DC5010%,STATUS%)
370   FG=VAL(LEFT$(RD$,LENGTH%))
380   PRINT DM,FG
390   AMPLITUDE = AMPLITUDE+.1
400 NEXT
410 END
```

This program is in file \UTILITY\PCTO-NEC.BAS on the disk.

```
10 '  Title        : pcto-nec
30 '  Purpose : to transfer data files from another
40 '  computer to the PC via the IEEE-488 bus.
80 '  Instructions: Run this program then send data
90 '     from the source computer to device address one.
100 '    The PC appears as a printer at address one.
160 '
170 DEFINT A-Z
180 LINE.COUNT = 0
190 PRINT "NOTE: you must set your PC-488 as a device"
200 PRINT "  not controller).  Set switch S1-8 ON."
220 PRINT "--- Hit ENTER to continue"
230 LINE INPUT R$
240 DEF SEG = &HCC00
250 INIT = 0 :XMIT = 3 :RECV = 6
260 R$=SPACE$(80)
270 PC.ADDRESS = 1 :DEVICE = 2    'pc gp-ib address 1
280 INPUT "Enter file name to save ",FILENAME$
290 IF FILENAME$= "" THEN STOP
300 OPEN FILENAME$ FOR OUTPUT AS #1
310 CALL INIT(PC.ADDRESS, DEVICE)
320 IF INP(&H2BC) AND 4 THEN 330 ELSE 320
330 CALL RECV  (R$,L,S)
340 R$ = LEFT$ (R$,L)          'trim the string to length
350 PRINT R$                   'print transferred line
360 PRINT #1,R$                ' save line to file
370 LINE.COUNT=LINE.COUNT+1
380 R$=SPACE$(80)
390 IF S<>8 THEN 330           'continue until timeout
410 CLOSE
420 PRINT LINE.COUNT;" lines transferred."
430 END
```

This program illustrates the use of the BASIC Interrupt Software driver available from CEC (part # 01000-10300) From DOS you type one command, GPIBINT, to install a GPIB interrupt driver in place of the light pen interrupt. After the interrupt driver is installed -

- ON PEN defines the subroutine that will run when a gp-ib interrupt is received.
- PEN ON enables the trapping of gp-ib interrupts.
- PEN OFF disables the trapping of gp-ib interrupts.
- PEN STOP logs gp-ib interrupts.
- PEN (n) functions return interrupt status information.

```
100 ' Title   :  SRQINT.BAS
120 ' Purpose : Demonstrates SRQ interrupt processing.
140 CLS
150 DEFINT A-Z
160 DEF SEG=&HCC00
170 INIT=0
180 MY.ADDR=20 : SYSCON=0
190 CALL INIT (MY.ADDR,SYSCON)
200 OUT &H2BA, &H40                'enable SRQ NEC 7210
230 OUT &H21,INP(&H21) AND &HF7     'enable int. channel
240 ON PEN GOSUB 310                'specify int. routine
250 LOCATE 10,20:INPUT "Press enter to enable ",NULL$
260 LOCATE 11,16:PRINT "Status displayed on SRQ"
270 PEN ON                      'turn on BASIC interrupts
280 ON TIMER(1) GOSUB 400       'specify timer subroutine
290 TIMER ON                    'turn on timer interrupt
300 GOTO 300                    'wait for interrupts
310 ' GP-IB Interrupt Subroutine
340 CLS
350 INT.REG0   = PEN(1)
360 INT.REG1   = PEN(2)
370 LOCATE 12,27 :PRINT "Int. reg. 0 "; HEX$(INT.REG0)
380 LOCATE 13,27 :PRINT "Int. reg. 1 "; HEX$(INT.REG1)
390 RETURN
400 ' Timer interrupt subroutine
430 LOCATE 8,35:PRINT TIME$
440 RETURN
```

SRQINT.BAS line-by-line explanation.

140 - 190: clear the screen, define the PC's gp-ib address and status as a system controller.

200 - 220: enable the SRQ interrupt by writing to the interrupt mask register on PC488. Any set of interrupt conditions can be enabled.

230: enable the 8259 so that it can recognize gp-ib interrupts.

240: after the interrupt handler is installed, ON PEN defines the subroutine that will run when a gp-ib interrupt is received.

270: gp-ib interrupts are now enabled.

280 - 290: a timer interrupt is serviced every second. These lines illustrate that timer interrupts, gp-ib interrupts, and other interrupts can all be serviced in one program.

300: wait for timer and gp-ib interrupts.

310 - 390: gp-ib interrupt service routine. PEN(1) and PEN(2) statements return the values of the interrupt status registers for display by the program.

400 - 440: timer interrupt service routine. Displays time-of-day updated every second.

```
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.
```

C Examples

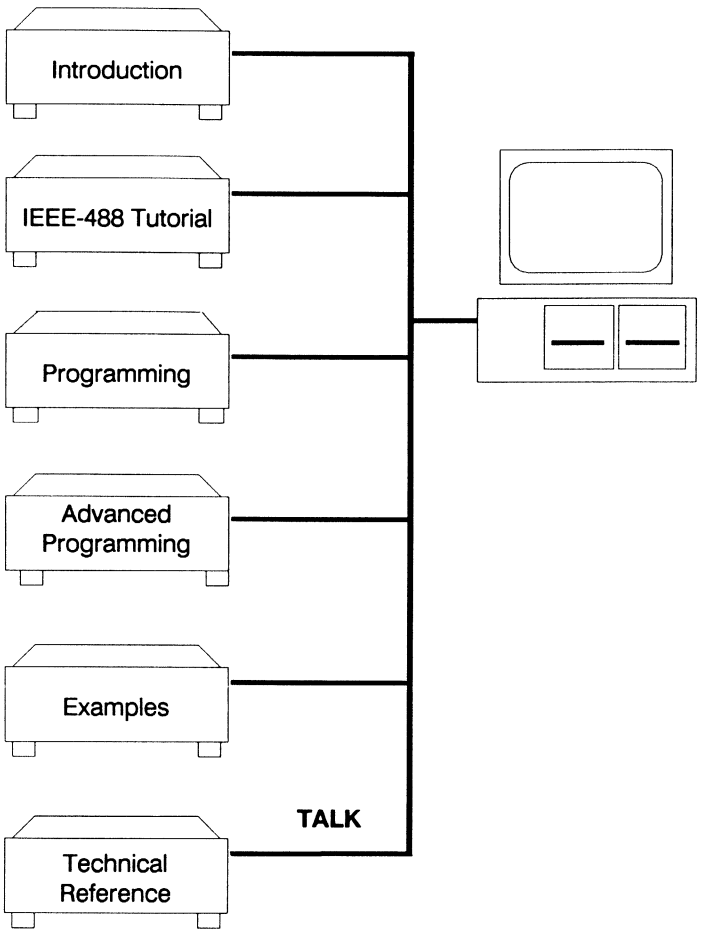
```
/*
  Example 1: use of SEND & ENTER to communicate
  with an instrument (Keithley 195 meter)
*/
#include <ieee-c.h>

#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

Introduction

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.

Electrical Specifications

The maximum 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 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 CEC-488 drivers (75160 and 75162) incorporate these improvements.

Driver specifications

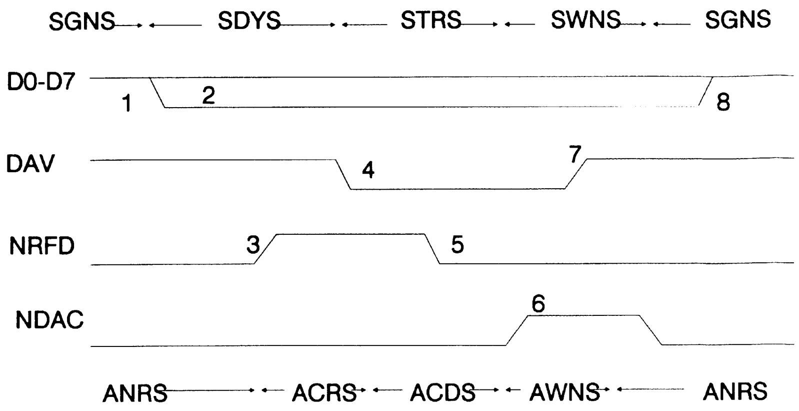
$V_{OL} < +0.5$ V at 48 mA continuous sink current
$V_{OH} > +2.4$ V at 5.2 mA continuous source current

Receiver specifications

Preferred	**Accepted**
$V_{OL} = V_{T-} >= +0.8$ volt	$V_{OL} <= +0.8$ volt
$V_{OH} = V_{T+} <= +2.0$ volt	$V_{OH} >= +2.0$ volt

Hysteresis: $V_{T+} - V_{T-} >= +0.4$V

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.

Mechanical Specifications

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 the CEC-488 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.

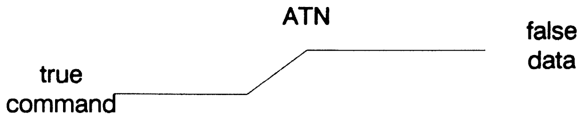
Bus Line Definitions

Data Bus

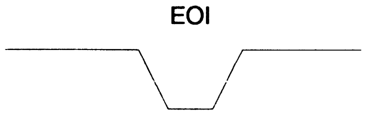
DIO1 through DIO8 - bidirectional asynchronous data lines.

Management Bus

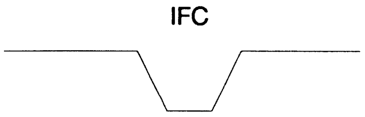
ATN (Attention) - a bus management line that indicates whether the current data is to be interpreted as data or a command. When asserted with EOI it indicates that a parallel poll is in process.



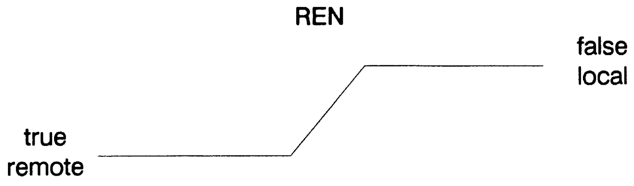
EOI (End or identify) - a bus management line that indicates the termination of a data transfer. When asserted with ATN, it indicates that a parallel poll is in process.



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.

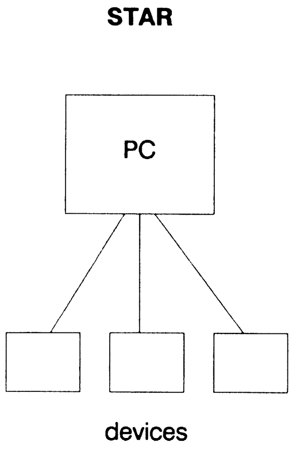
Transfer Bus

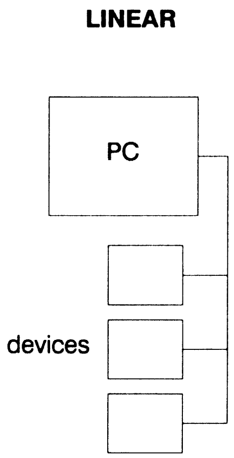
DAV (Data Valid) - one of three handshake lines used to indicate availablility and validity of information on the DIO lines.

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.





APPENDICES

Well mack the Finger said to Louie the King,
"I got forty red, white and blue shoe strings
and a thousand telephones that don't ring.
Do you know where I can get rid of these things?"

- Bob Dylan from Highway 61 Revisited
Reprinted by permission of Bob Dylan
Copyright 1965 Warner Brothers

BASIC Language Interface

This section documents the use of BASICA or GWBASIC with the CEC IEEE-488 interfaces. BASIC uses IEEE-488 software which resides in memory on the interface card.

On PS488, this memory-resident software must be loaded. You need to run the file "PS488.EXE" from the application disk.

IEEE-488 routines are called in BASIC and BASICA with the CALL statement. Before CALL can be used, DEF SEG must be used to indicate the memory segment address of the interface. A variable named for each IEEE-488 routine must be set to the offset for that routine (example: INITIALIZE = 0).

Variables used with CEC-488 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=&HCC00
20 INITIALIZE=0
```

The address in line 10 must match the memory address of the interface board. Subroutine offset variables, as in line 20, only need to be set once in the program.

IEEE-488 Subroutine calls

The following pages give the calling information for each CEC-488 routine.

INITIALIZE

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

- address% is the GPIB address to be used by CEC-488. It must be in the range 0 to 30.
- level% indicates whether CEC-488 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%)
```

- address% 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").
- 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

```
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.

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.

DMA2

The DMA2 routine is used when DMA (direct memory access) transfers are desired. In other languages, DMA is used when TARRAY or RARRAY is called and the DMA channel has been set with the DMACHANNEL call. In BASIC, DMA must be started explicitly with the DMA2 call.

```
DMA2 = 206
CALL DMA2 (seg%,ofs%,count%,mode%,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 transferred.
- mode% indicates the DMA mode settings to be used.
  A value of &H2109 will cause DMA output to occur.
  A value of &H2105 will cause DMA input to occur.
- status% indicates whether the transfer went OK.

SRQ, SETPORT, BOARDSELECT, DMACHANNEL, SETTIMEOUT, SETOUTPUTEOS, SETINPUTEOS

These functions are NOT supported with the BASIC interpreter. Microsoft QuickBASIC, however, supports the full set of IEEE-488 functions.

Examples

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

```
10 DEF SEG=&HC400
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:

```
10 DIM D%(1000)
20 DEF SEG=&HC400
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 Language Interface

All versions of the Microsoft QuickBASIC compiler are supported. At the time this manual was written, the interface had been tested up to version 4.5 of QuickBASIC. It is expected that future versions of QuickBASIC should work without changes.

Interface files

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

For QuickBASIC 1.0 you will need:
    \QB\IEEEQB3.OBJ
    \IEEE488.LIB

For QuickBASIC 2.0 or 3.0 you will need:
    \QB\IEEEQB3.EXE

For QuickBASIC 4.0 and later you will need:
    \QB\IEEEQB.BI
    \QB\IEEEQB.QLB
    \QB\IEEEQB.LIB

Compiling programs

First, write your QuickBASIC program using the IEEE-488 subroutine calls shown in the next section. Then, to compile and run your program:

For QuickBASIC 1.0 you compile your program normally and link with the supplied object module and library:

```
C> BASCOM myprog;
C> LINK myprog ieeeqb3,,,ieee488;
```

For QuickBASIC 2.0 or 3.0, you can load the supplied user library when you run QuickBASIC. The IEEE-488 subroutines will then be available:

```
C> QB /L ieeeqb3.exe
```

If you are using version 2.0 or 3.0 and wish to compile and link from the DOS command line and not use the QuickBASIC environment, you can link your program with the same object module used for QuickBASIC 1.0.

For QuickBASIC 4.0 and later, make sure you include the file IEEEQB.BI in your program (see the examples later). To load the supplied user library:

```
C> QB /L ieeeqb.qlb
```

IEEE-488 Subroutine calls

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

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

- "r$" is the string into which the received data will be placed. Space for the received data must be allocated, usually with the SPACE$ 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.
- "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.
- "status%" indicates the success or failure of the operation.

RECEIVE

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

- "r$" is the string into which the received data will be placed. Space for the received data must be allocated, usually with the SPACE$ 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.
- "l%" is a variable which will be set to the actual received length.
- "status%" indicates the success or failure of the data transfer.

TARRAY

For QuickBASIC 2.0 or 3.0

```
CALL PTR86(s%,o%,VARPTR(d%(1))
CALL TARRAY (s%,o%,count%,eoi%,status%)
```

- "d%" is the array variable containing the data to be transmitted. "s%" and "o%" are set to the segment and offset parts of the memory address by the PTR86 routine, supplied with QuickBASIC.
- "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.

For QuickBASIC 4.0 or later

```
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

For QuickBASIC 2.0 or 3.0

```
CALL PTR86(s%,o%,VARPTR(d%(1))
CALL RARRAY (s%,o%,count%,l%,status%)
```

- "d%" is the array variable into which data will be received. "s%" and "o%" are set to the segment and offset parts of the memory address by the PTR86 routine, supplied with QuickBASIC.
- "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.

For QuickBASIC 4.0 or later

```
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%

The SRQ% function is supported ONLY in QuickBASIC 4.0 and later!

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

OR

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

SETPORT

Note: this routine is not needed on PS488, 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.

```
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.

Examples

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

```
' $INCLUDE: 'ieeeqb.bi' '(for version 4.x)
CALL INITIALIZE (21,0)
CALL SEND (16,"F0R0X",status%)
r$ = 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%)
```

Turbo Pascal Language Interface

This manual describes the use of Borland's Turbo Pascal with CEC's IEEE-488 interfaces.

Interface files

For Turbo Pascal 4.0 you will need to copy:

```
C> copy A:\turbopas\ieeepas4.tpu C:ieeepas.tpu
```

For Turbo Pascal 5.0 you will need to copy:

```
C> copy A:\turbopas\ieeepas.tpu C:
```

At the time this manual was written, version 5.0 was the most recent revision to Turbo Pascal. It is expected that future versions will work with the same interface software. If necessary, you can re-build the Turbo Pascal UNIT file from the source files IEEEPAS.PAS and PAS488.OBJ.

If you have Turbo Pascal version 3.0 or earlier, look in the file "\TURBOPAS\OLD\TURBOPAS.DOC" on the IEEE-488 applications disk.

Compiling programs

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

```
USES ieeepas;
```

Compile and run your program normally.

IEEE-488 Subroutine calls

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,l,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.
- "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 PS488, 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.

```
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.

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.
```

This manual describes the use of C with CEC's IEEE-488 interfaces. This software has been tested with Microsoft C, version 3.0 and later and with Borland's Turbo C. The same software should work with other C compilers, as long as they support the keywords "pascal" and "far".

Interface files

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

    \C\IEEE-C.H
    \IEEE488.LIB

If you are writing a C program to run under the OS/2 operating system, you will need these files:

    \C\IEEE-C.H
    \OS2\IEEEOS2.LIB
    \OS2\IEEEOS2.DLL

The IEEEOS2.LIB file should be linked with your program in place of the IEEE488.LIB file. IEEEOS2.DLL must be installed on your OS/2 system in order to run your program.

Compiling programs

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 and link it with the library IEEE488.LIB.

Microsoft C

```
C> CL myprog.c /link ieee488
```

Turbo C

For Borland's Turbo C, make a project file listing both your C source file and the required library. See the Turbo C manual for more information on project files. For example, create the file MYPROG.PRJ with this text:

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

Then, select this file as the current project using the Project menu. Compile and run normally (Alt-R).

IEEE-488 Subroutine calls

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.
- "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" (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" (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 PS488, 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.

```
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.

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);
}
```

FORTRAN Language Interface

This manual describes the use of FORTRAN with CEC's IEEE-488 interfaces.

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
```

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;
```

IEEE-488 Subroutine calls

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.

In Microsoft FORTRAN all strings constants passed to FORTRAN routines should be C-style strings, which may be created by placing the letter C just after the closing quote of the string constant. In RM/FORTRAN, this is not necessary.

For example, you can call SEND as either:

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

or

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

Note also that integer constants do not contain a decimal point.

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 PS488, 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.

```
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.

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,'FOROX'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

CEC-488 comes with support for many popular programming languages. If you wish to interface directly with the CEC-488 software library from Macro Assembler code or from an unsupported language, this section describes the necessary steps.

CEC-488 software is provided as a Microsoft format library in file IEEE488.LIB. 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\ROM-CALL.DOC on the CEC-488 applications disk.

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 CEC has 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 or IBM Macro Assembler, simply use the commands:

```
C> MASM myprog;
C> LINK myprog,,,ieee488;
```

IEEE-488 Subroutine calls

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 CEC-488 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.

```
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.

```
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.

```
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

Note: this routine is not needed on PS488, 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.

```
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
```

(continued...)

```
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
     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
```

CEC-488 includes a library of interface routines which work with the Microsoft 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

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.

BASIC

Copy the file IEEEOS2.LIB from the application disk into IEEE488.LIB on your system disk (changing the name of the file).

Compile your BASIC code to create an .OBJ file.

Link the object file from BASIC with IEEEQB.OBJ and IEEE488.LIB.

Microsoft FORTRAN

Copy the file IEEEOS2.LIB from the application disk into IEEE488.LIB on your system disk (changing the name of the file).

Compile your FORTRAN code to create an .OBJ file.

Link the object file from FORTRAN with IEEEFOR.OBJ and IEEE488.LIB.

Troubleshooting

This section describes solutions to various problems that can occur when conflicts exist among hardware devices in the PC, or among software drivers.

The sections which follow cover each of CEC's IEEE-488 boards.

For information on hardware switch settings, see the appendix on Hardware Configuration.

›› Computer fails to boot correctly.

- There is a conflict between PC< >488 and other hardware in your PC. It may be:
  - A memory conflict, usually with a display adapter, network adapter, or expanded memory board. Try a different memory address setting on switch S1, such as location D000 or C800.
  - A DMA conflict, usually with a network adapter or data acquisition adapter. Try disabling DMA on PC< >488 by removing the two jumpers on J4. If this works, you can either avoid using DMA on PC< >488 or try DMA channel 3 to see if it is available.
  - An I/O conflict. This is very unusual. The only hardware reported to conflict with PC< >488's I/O address is the SCSI disk adapter on the Qualogy AT system board. If there is an I/O conflict, set switch S2 to another address, such as 2A8, and see the SETPORT routine in the Programming section.

›› Computer hangs when a BASIC program is run.

- The BASIC program probably has the wrong address setting.
  - There is a line like "DEF SEG=&HCC00" in the BASIC program. This line must match the address set on switch S1.

›› 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 PC< >488. If these work, double-check your code and look again at the example programs in the language interface appendices 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 connec-

tor 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 fails to boot once 4x488 is installed.

- This could be a DMA conflict.
  - Boot from a floppy disk and run INSTALL again. Tell INSTALL to disable DMA.
- There could be a hardware conflict which cannot be recognized by the INSTALL program.
  - Try removing as many add-in boards as possible from your PC.
- There could be a software problem. Does the computer begin to boot and print messages on the screen and then crash?
  - Try removing other device drivers from CONFIG.SYS, and other programs from AUTOEXEC.BAT.

›› Serial port doesn't work.

- This could be a confusion over port numbering.
  - Run INSTALL again and note the ports shown as already present in your PC as well as those provided by 4x488. Some IBM serial/parallel port boards label the settings backwards, causing confusion. If you had such a board, set to COM2, DOS would still have referred to it all along as COM1, since it was the only port present. When 4x488 is added, however, 4x488 becomes COM1. You can either use the cards this way or switch the other serial port card to COM1 and run INSTALL again.
- It could be a cabling problem.
  - If possible, try the same device and cable on another serial port, either on this computer or another computer. Make sure that the switch on the back panel of 4x488 is in the UP position so that it matches a standard PC/AT serial port.
- It could be an interrupt problem.
  - If 4x488 is acting as COM1, it should be set for interrupt level 4. If it is COM2, it should be interrupt level 3. 4x488 comes from the factory set for interrupt level 3.
- It could be a board problem.
  - Try running the EMMTEST program provided with 4x488. This tests the serial port as well as other hardware on the board.

›› Expanded memory does not work.

- It may be a conflict with another expanded memory card.
  - Only one brand of expanded memory can exist at one time. Set either 4x488 or the other card to EXTENDED memory mode.

- You may have a PC with incorrect timing (in high-speed mode).
  - The Leading Edge Model D2 can have this problem. Run the EMMTEST program, which will tell you if the memory is OK, but expanded memory mode does not work.
- It may be a RAM problem.
  - Run the EMMTEST program to test the board.

›› Computer hangs when a BASIC IEEE-488 program is run.

- The BASIC program probably has the wrong address setting.
  - There is a line like "DEF SEG=&HCC00" in the BASIC program. This line must match the address set when INSTALL was run (except that the last zero is left off).

›› 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 4x488. If these work, double-check your code and look again at the example programs in the language interface appendices 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 IEEE-488 program is run.

- The software is probably not loaded on the board.
  - You must run the PS488 program to load the software.
- The BASIC program probably has the wrong address setting.
  - There is a line like "DEF SEG=&HCC00" in the BASIC program. This line must match the address set when INSTALL was run (except that the last zero is left off).

» 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 PS< >488. If these work, double-check your code and look again at the example programs in the language interface appendices 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 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.

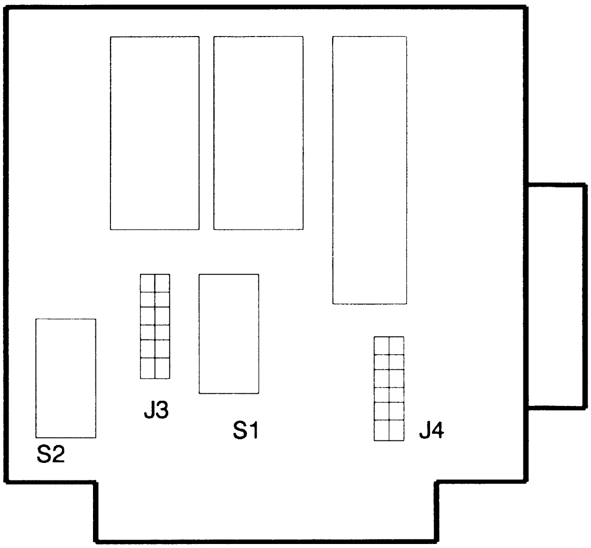
CEC manufactures three models of IEEE-488 interface: PC< >488, 4x488, and PS< >488. The hardware configuration options of each board are covered in the following sections of this appendix.

Compatibility

PC< >488 operates in any IBM PC, XT, AT, the IBM PS/2 model 30 and below, or any PC compatible. It has been in use since 1983 in thousands of different PC's worldwide.

4x488 operates in any IBM AT or compatible. Because it provides high-speed 16-bit memory, 4x488 is carefully designed to make the best possible use of the available AT hardware cycle time. There are some AT compatibles which do not provide correct memory timing (usually in an attempt to speed up the computer). These compatibles will work with 4x488 in extended memory mode, but not in expanded memory mode. The EMMTEST program provided with 4x488 can check whether this problem is present on your machine. Our experience has been that very few compatibles have this problem.

PS< >488 operates in any Micro Channel™ computer, which includes the IBM PS/2 model 50 and above, as well as compatibles like the Tandy 5000MC.



Note: there is an older hardware version of PC< >488 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 CEC and request an older manual and software disk to go with the TI card.

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.

First, remove PC488 and run the PCMAP program from the application disk. Unused memory ranges in your computer will be listed with the label "empty". Choose an unused memory area and set the switches as shown below:

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 8 on switch S1 indicates whether or not the PC< >488 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 in the Programming section of the manual.

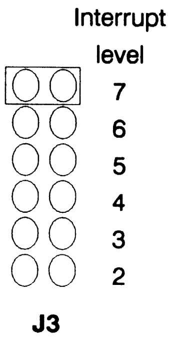
Some options for switch S2 are shown below:

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

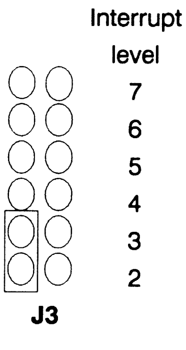
Interrupt level (J3)

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:

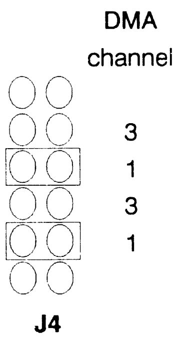


DMA channel (J4)

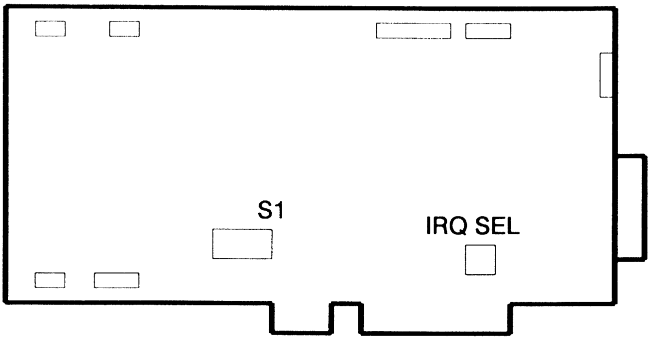
Jumper block J4 controls the DMA (direct memory access channel) used by PC488. DMA is used for high speed data transfers (see the DMACHANNEL subroutine in the Programming section of the manual).

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.



Most settings on 4x488 are done through the INSTALL software. This includes the IEEE-488 firmware memory address, the serial port I/O address, the parallel port address, and the memory configuration. Those settings that are available as hardware options are shown in this section. If you have difficulty with the software-controlled configuration process, see the appendix on Troubleshooting.

I/O Address Switch (S1)

Switch S1 on 4x488 controls the I/O addresses for 4x488 and 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 for the GPIB port, see the SETPORT subroutine in the Programming section of the manual.

Some options for switch S1 are shown below:

Position				I/O Address
1	2	3	4	
on	on	on	on	208
off	on	on	on	218
on	off	on	on	228
off	off	on	on	238
on	on	off	on	248
off	on	off	on	258
on	off	off	on	268
on	off	on	off	2A8
off	off	on	off	2B8

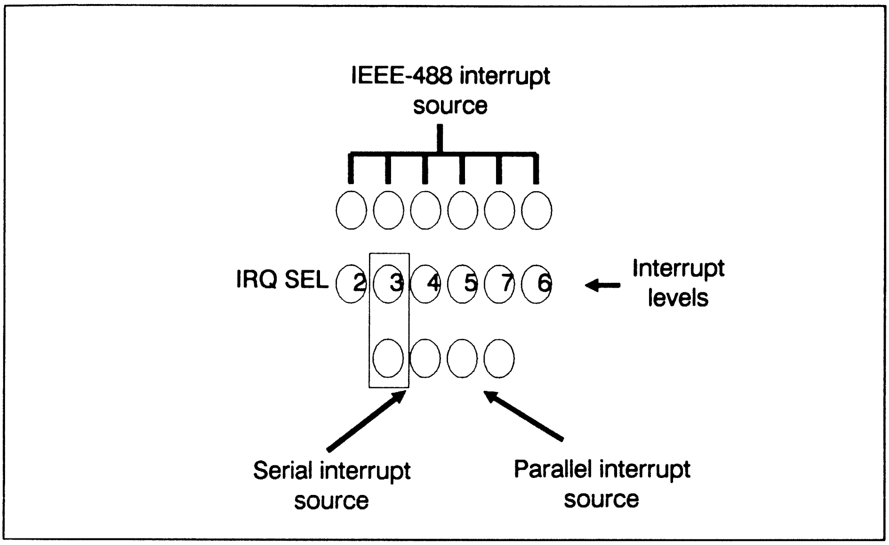
Positions 1 through 4 control the I/O address for the 4x488 expanded memory control port. Positions 5 through 8 control the I/O address for the GPIB port. Positions 5 through 8 use the same patterns as 1 through 4, shown above.

The factory default settings are 258 for the 4x488 port and 2B8 for the GPIB port.

Interrupt level (IRQ SEL)

Jumper block IRQ SEL controls the interrupt levels used by 4x488. Most IEEE-488 applications do not require interrupts. Serial ports usually DO require interrupts. 4x488 is shipped from the factory with interrupt level 3 enabled for the serial port. This is the usual setting for COM2. If you set 4x488 for COM1, you will want to change the interrupt jumper to level 4.

You may set the interrupt jumper as shown below:



Simply place interrupt jumper blocks so as to connect the center row with one of the outside rows. The jumper shown in the diagram, for example, connects interrupt level 3 to the serial port interrupt.

DMA channel

DMA is used for high speed data transfers (see the DMACHANNEL subroutine in the Programming section of the manual).

4x488 uses 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 disable DMA on 4x488. DMA is not required - it simply makes faster transfers possible. DMA may be disabled with the INSTALL program.

Serial port

The serial port pinout is as follows:

DB9	DB25	Signal	Definition
3	3	TxD	transmitted data
2	2	RxD	received data
7	4	RTS	request to send
8	5	CTS	clear to send
6	6	DSR	data set ready
5	7	ground	signal ground
1	8	DCD	data carrier detect
4	20	DTR	data terminal ready
9	22	RI	ring indicator

The DB9 9-pin connector is used on 4x488. A 9 to 25 pin adapter cable is available.

Switch SW2, on the back panel of 4x488, controls the receive and transmit pins on the serial port. Connecting some devices to the serial port may require reversing these two pins. Switch SW2 allows you to do this without special cables. SW2 is normally up, for what is called DTE operation (pin 3 is transmit, pin 2 is receive). When SW2 is down, it in in DCE mode (pin 3 is receive, pin 2 is transmit).

Some devices make use of the CTS, DSR, and/or DCD pins to indicate their readiness to transmit or receive data. If these lines are required, and the device is a DTE-type device, special cabling may be needed. However, if you wish to have the computer ignore these ready, or "handshaking" pins, you can run the program CECRS232, provided on the 4x488 Installation disk.

The CECRS232 program can make any of the three pins CTS, DSR, or DCD appear to always be TRUE as far as the computer is concerned. The default configuration for 4x488 has DCD always TRUE, and the other two signals passed through as they actually appear on the cable. You can run CECRS232 as follows:

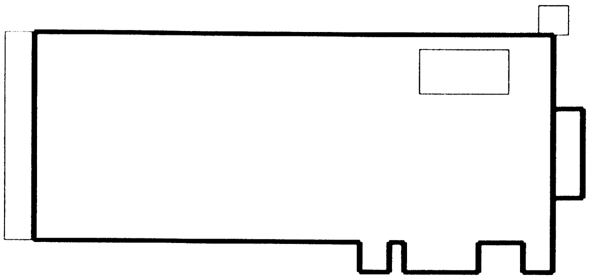
```
A> CECRS232 bdnum CTS DSR DCD
```

where "bdnum" is a number from 1 to the number of 4x488's installed. If bdnum is not specified, one is assumed. "CTS", "DSR", and "DCD" are all optional. Include only those signals you wish to be seen as always TRUE.

Parallel port

The parallel printer port pinout is as follows:

Pin	Signal
1	-STROBE
2 to 9	DATA
10	-ACK
11	+BUSY
12	+PE
13	+SELECT
14	-AUTO FD XT
15	-ERROR
16	-INIT
17	-SELECT IN
18 to 25	GROUND



PS< >488 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 PS< >488, you should boot your computer from the Reference diskette and choose "Set configuration", then "Change configuration". PS< >488 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 PS< >488 (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

4x488 can contain up to 4 megabytes of RAM. Either of two types of dynamic RAM chip can be used in 4x488:

- 256K bit, 120 nsec or faster DRAMs (Hitachi HM50256-12 or equivalent)
- 1M bit, 120 nsec or faster DRAMs (Toshiba TC511000P-12 or equivalent)

The table below shows the amount of memory available with each configuration:

Type of chips	# of columns	Amount of RAM (bytes)
256K	2	512K
256K	4	1 meg
1M	2	2 meg
1M	4	4 meg

If only two columns of memory are to be installed, they must be columns 0 and 2.

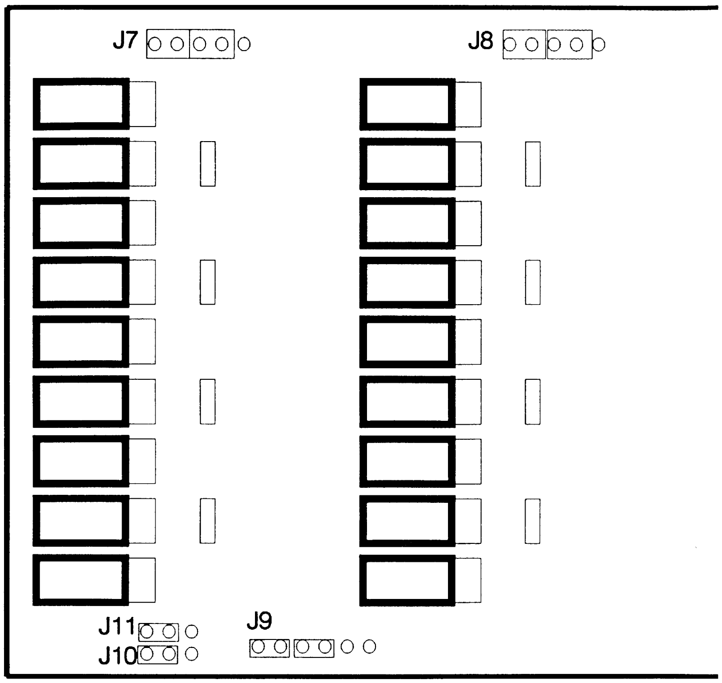
The RAM sockets on the 4x488 board have 20 pin locations each. If 256K chips are used, they **must** be placed all the way to the left in the sockets. If 1M chips are used, they **must** be placed all the way to the right in the sockets. Jumpers J7, J8, J9, J10, and J11 must also be placed correctly for the type of RAM chip being used. The columns of capacitors between the rows of RAM chips must also be moved depending on the type of RAM chip.

This is very important!
If jumpers J7 through J11 do not match the RAM chip type, damage to the chips may occur!

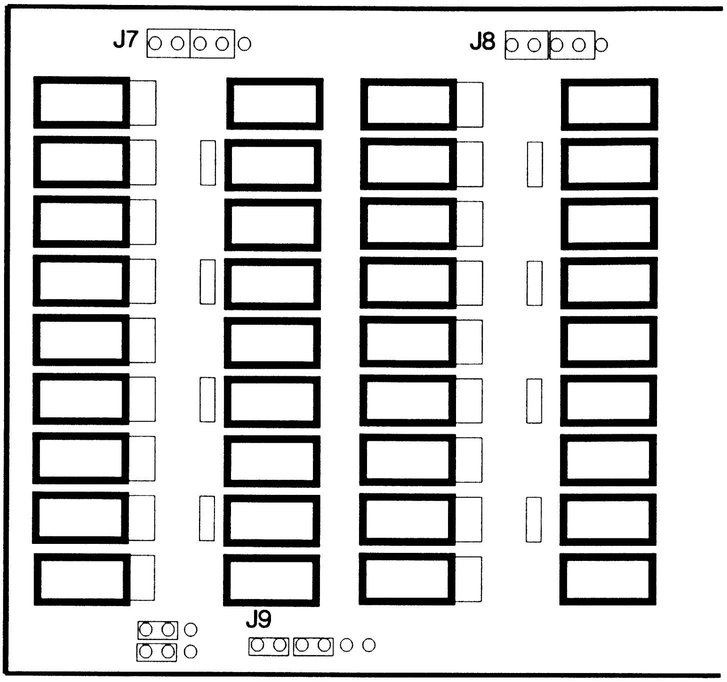
Note: the correct locations for all these jumpers and capacitors for use with 256K chips are shown on the board as empty rectangle outlines. The correct locations for use with 1M chips are shown as filled-in rectangles.

CEC factory configures 4x488 by wiring the pins on the jumpers together. You can simply remove the wires with needlenose pliers and add jumpers in the new positions, or you can have CEC factory re-configure the board for you.

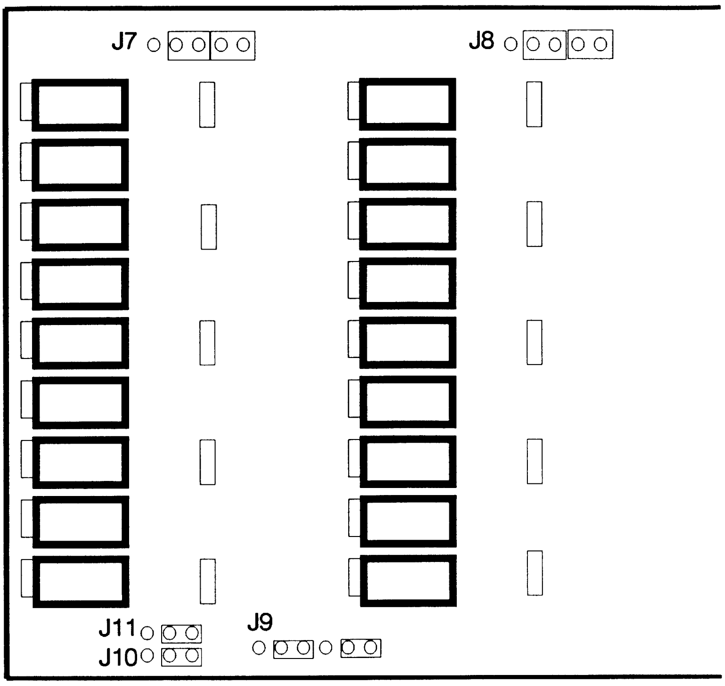
The illustrations on the following pages show how to insert the chips for each memory configuration.



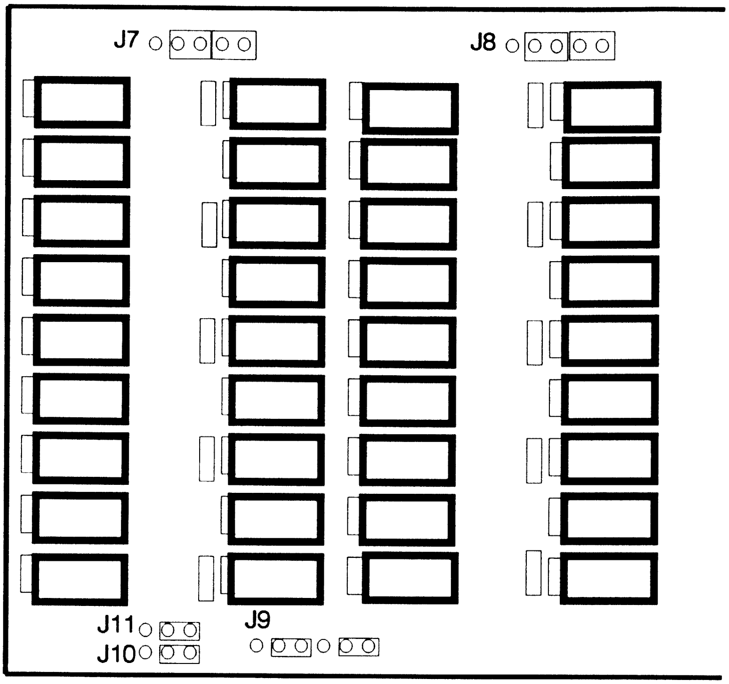
512K bytes of memory: 2 rows of 256K chips



1024K bytes of memory: 4 rows of 256K chips



2M bytes of memory: 2 rows of 1M chips



4M bytes of memory: 4 rows of 1M chips

ASCII Char.	Binary	Hex	Decimal	GPIB
NULL	00000000	00	0	
SOH	00000001	01	1	GTL
STX	00000010	02	2	
ETX	00000011	03	3	
EOT	00000100	04	4	SDC
ENQ	00000101	05	5	PPC
ACK	00000110	06	6	
BELL	00000111	07	7	
BS	00001000	08	8	GET
HT	00001001	09	9	TCT
LF	00001010	0A	10	
VT	00001011	0B	11	
FF	00001100	0C	12	
CR	00001101	0D	13	
SO	00001110	0E	14	
SI	00001111	0F	15	
DLE	00010000	10	16	
DC1	00010001	11	17	LLO
DC2	00010010	12	18	
DC3	00010011	13	19	
DC4	00010100	14	20	DCL
NAK	00010101	15	21	PPU
SYNC	00010110	16	22	
ETB	00010111	17	23	
CAN	00011000	18	24	SPE
EM	00011001	19	25	SPD
SUB	00011010	1A	26	
ESC	00011011	1B	27	
FS	00011100	1C	28	
GS	00011101	1D	29	
RS	00011110	1E	30	
US	00011111	1F	31	

ASCII Char.	Binary	Hex	Decimal	GPIB
space	00100000	20	32	LA0
!	00100001	21	33	LA1
"	00100010	22	34	LA2
#	00100011	23	35	LA3
$	00100100	24	36	LA4
%	00100101	25	37	LA5
&	00100110	26	38	LA6
’	00100111	27	39	LA7
(	00101000	28	40	LA8
)	00101001	29	41	LA9
*	00101010	2A	42	LA10
+	00101011	2B	43	LA11
,	00101100	2C	44	LA12
-	00101101	2D	45	LA13
.	00101110	2E	46	LA14
/	00101111	2F	47	LA15
0	00110000	30	48	LA16
1	00110001	31	49	LA17
2	00110010	32	50	LA18
3	00110011	33	51	LA19
4	00110100	34	52	LA20
5	00110101	35	53	LA21
6	00110110	36	54	LA22
7	00110111	37	55	LA23
8	00111000	38	56	LA24
9	00111001	39	57	LA25
:	00111010	3A	58	LA26
;	00111011	3B	59	LA27
<	00111100	3C	60	LA28
=	00111101	3D	61	LA29
	00111110	3E	62	LA30
?	00111111	3F	63	UNL

ASCII Char.	Binary	Hex	Decimal	GPIB
@	01000000	40	64	TA0
A	01000001	41	65	TA1
B	01000010	42	66	TA2
C	01000011	43	67	TA3
D	01000100	44	68	TA4
E	01000101	45	69	TA5
F	01000110	46	70	TA6
G	01000111	47	71	TA7
H	01001000	48	72	TA8
I	01001001	49	73	TA9
J	01001010	4A	74	TA10
K	01001011	4B	75	TA11
L	01001100	4C	76	TA12
M	01001101	4D	77	TA13
N	01001110	4E	78	TA14
O	01001111	4F	79	TA15
P	01010000	50	80	TA16
Q	01010001	51	81	TA17
R	01010010	52	82	TA18
S	01010011	53	83	TA19
T	01010100	54	84	TA20
U	01010101	55	85	TA21
V	01010110	56	86	TA22
W	01010111	57	87	TA23
X	01011000	58	88	TA24
Y	01011001	59	89	TA25
Z	01011010	5A	90	TA26
[	01011011	5B	91	TA27
\	01011100	5C	92	TA28
]	01011101	5D	93	TA29
^	01011110	5E	94	TA30
_	01011111	5F	95	UNT

ASCII Char.	Binary	Hex	Decimal	GPIB
‘	01100000	60	96	SC0
a	01100001	61	97	SC1
b	01100010	62	98	SC2
c	01100011	63	99	SC3
d	01100100	64	100	SC4
e	01100101	65	101	SC5
f	01100110	66	102	SC6
g	01100111	67	103	SC7
h	01101000	68	104	SC8
i	01101001	69	105	SC9
j	01101010	6A	106	SC10
k	01101011	6B	107	SC11
l	01101100	6C	108	SC12
m	01101101	6D	109	SC13
n	01101110	6E	110	SC14
o	01101111	6F	111	SC15
p	01110000	70	112	SC16
q	01110001	71	113	SC17
r	01110010	72	114	SC18
s	01110011	73	115	SC19
t	01110100	74	116	SC20
u	01110101	75	117	SC21
v	01110110	76	118	SC22
w	01110111	77	119	SC23
x	01111000	78	120	SC24
y	01111001	79	121	SC25
z	01111010	7A	122	SC26
{	01111011	7B	123	SC27
|	01111100	7C	124	SC28
}	01111101	7D	125	SC29
~	01111110	7E	126	SC30
DEL	01111111	7F	127	SC31

Using PRINT and INPUT for GPIB control

The simplest approach to controlling GPIB instruments with CEC-488 makes use of the standard PRINT and INPUT statements. For example, to initialize a Tektronix DM5010 at GPIB address 20:

```
  10 OPEN "gpib20" FOR OUTPUT AS #1
  20 PRINT #1,"INIT"
```

This method of programming can be used whenever data transfer speed is not an important issue.

To set CEC-488 up for use with PRINT and INPUT statements, you should install the DOS device driver "CECGPIB.BIN" from the CEC-488 applications disk. Copy \UTILITY\CECGPIB.BIN to the system disk you use to boot your computer. Then, add the following line to your CONFIG.SYS file on the system disk:

```
DEVICE=CECGPIB.BIN 21
```

where the value 21 specifies the GPIB address to be used by CEC-488, 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 the CEC-488 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 BASIC:

```
10 OPEN "gpib4" FOR OUTPUT AS #1
20 OPEN "gpib4" FOR INPUT AS #2
30 DEF SEG=&HC400  ' set up for firmware CALLs
40 SPOLL=12
50 '
60 PRINT #1,"MEASURE"  ' send command
65 '
67 ADDRESS%=4 ' serial poll
70 CALL SPOLL (ADDRESS%,POLL%,STATUS%)
80 IF POLL%<>97 THEN 70
85 '
90 INPUT #2,VOLTAGE ' read value
100 '
110 PRINT VOLTAGE
120 '
130 CLOSE
140 END
```

When mixing firmware 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.

Quick Pascal Language Interface

This manual describes the use of Microsoft's Quick Pascal with CEC's IEEE-488 interfaces.

Interface files

For Quick Pascal you will need to copy:

```
C> copy A:\qp\ieeepas.qpu C:
```

At the time this manual was written, version 1.0 was the most recent revision to Quick Pascal. It is expected that future versions will work with the same interface software. If necessary, you can re-build the Quick Pascal UNIT file from the source files IEEEPAS.PAS and QP488.OBJ.

Compiling programs

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

```
USES ieeepas;
```

Compile and run your program normally.

IEEE-488 Subroutine calls

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,l,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.
- "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 PS488, 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.

```
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.

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,'FOROX',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.
```

CEC 488 interfaces come with a DOS device driver which allows you to use Hewlett-Packard style commands to control instruments. This driver can be used with the Labtech Notebook software package, or with your own programs in BASIC, C, FORTRAN, Pascal, or any other language.

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 CEC 488 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:

```
10 OPEN "IEEE" FOR OUTPUT AS #1
20 PRINT #1,"OUTPUT 16;INIT"
```

You can PRINT commands to the driver, and use INPUT to get data back. For example:

```
10 OPEN "IEEE" FOR OUTPUT AS #1
20 OPEN "IEEE" FOR INPUT AS #2
30 PRINT #1,"OUTPUT 12;READ"
40 PRINT #1,"ENTER 12"
50 INPUT #2,VOLTAGE
60 PRINT "The voltage is ";VOLTAGE
70 CLOSE
```

If you have multiple CEC 488 boards, and have loaded multiple drivers, the driver file names are "IEEE", "IEEE2", "IEEE3", and "IEEE4".

Converting HP computer programs to use CECHP

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:

```
100 TRIGGER 705
```

becomes this when using IBM BASICA or GWBASIC:

```
100 PRINT #1,"TRIGGER 5"
```

Reading data

On an HP computer with built-in HPIB control, you can use a statement like:

```
100 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:

```
100 PRINT #1,"ENTER 5"
110 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.

Commands

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 CEC 488 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, 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.

The same programs given above will work in QuickBASIC.

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
```