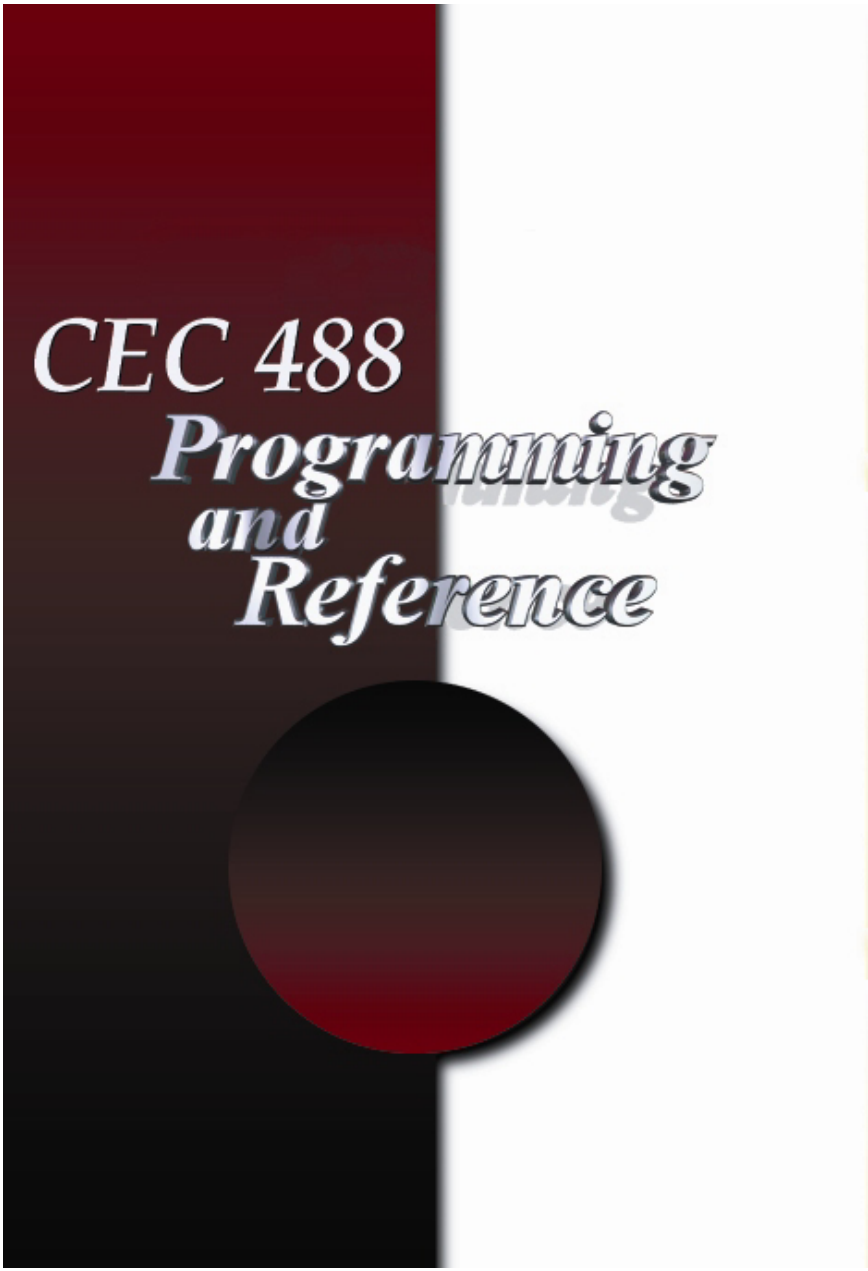

CEC 488
Programming
and
Reference

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

Part number 370966B-01
August 2004

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

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

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

Table of Contents

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

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

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

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

Your board can be used with your own programs in any of the popular programming languages for instrument control applications. It also works with many development environments such as LabView and TestPoint.

Your board's key features include:

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

- High-speed data transfers.

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

Installation

Installation of your CEC488 hardware and software is covered in the separate Installation Guide, which may be found printed on the CD-ROM insert and also in PDF format on the CD-ROM.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

New languages may be available on the applications CD-ROM that are not listed in the manual.

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

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

You will find the language interface files you need in various subdirectories under the installation directory you chose when installing the CEC488 software (usually this is **C:\Program Files\CEC488**).

Visual BASIC (including Visual BASIC for Application)

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

For VB.NET, the instructions are the same, but you need to use the **IEEEVB.VB** file you will find in the VB\VBNET subdirectory.

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

C or C++ (any vendor)

You will need the file **IEEE-C.H** (from the C subdirectory). You may want to copy this file to the working directory you will be using to develop your program.

Put the following code line at the top of your program:

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

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

for 32-bit Windows programs, use: IEEE_32M.LIB (for Microsoft), or IEEE_32B.LIB (for Borland)

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

Borland Delphi

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

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

```
uses ieeedel;
```

Microsoft C#

Add the file CEC488.cs to your C# .NET project. This file defines the CEC488 class,w hich contains all the interface routines as public methods.

Other languages

See the specific language-interface appendices in the latter section of this manual, or check the README file installed as part of the CEC488 software.

Many DOS-specific languages are supported only with CEC488 v7.0 and earlier. If you need to obtain this version of software, check our web site at www.cec488.com.

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

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

INITIALIZE (my.address,level)

where:

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

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

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

Visual BASIC:

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

Delphi:

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

C or 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 **Send** routine makes this easy:

SEND (address,info,status)

where:

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

This is the first time the **status** argument has appeared. It is generally the last argument in all board calls. Status will return with the value zero if the transfer went okay. The most common nonzero 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.

Visual BASIC:

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

Delphi:

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

C or C++:

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

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

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

However, some devices contain submodules with **secondary** addresses.

You can access these devices by combining the primary and secondary addresses by the following formula:

100 * primary + secondary

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

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

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

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

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

where:

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

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

Visual BASIC:

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

Delphi:

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

C or C++:

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

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

Visual BASIC:

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

Delphi:

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

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

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

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

C or C++:

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

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

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

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

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

C#:

```
Private void button1_Click(object sender,
System.eventArgs e)
{
      int status,len;
      String r;

      // Initialize the GPIB system
      CEC488.initialize (21,0);

      // Send a command  (take a measurement)
      CEC488.send (7,"MEASURE", out status);

      // Read the measured value
      CEC488.enter ( out r,30, out len,7, out status);

      textBox1.Text = "Value is " + r;
}
```

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

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

Visual BASIC:

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

Delphi:

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

C or C++:

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

A serial poll reads the status of a single device.

SPOLL (address,poll,status)

where:

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

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

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

Visual BASIC:

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

Delphi:

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

C or C++:

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

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

PPOLL (poll)

where:

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

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

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

Visual BASIC:

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

Delphi:

```
ppoll (poll);
```

C or C++:

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

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

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

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

ListenerPresent (addr)

where:

- addr is the device address you want to check

returns: 0 if the device is not present, nonzero if it is present.

Visual BASIC example

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

Delphi example:

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

C or C++:

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

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

Testing for the Presence of the GPIB Board

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

GPIBBoardPresent

returns: 0 if no board is present, nonzero if board is installed

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

Visual BASIC:

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

Delphi:

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

C or C++:

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

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

GPIBFeature (feature)

where

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

Feature	number	Description
IEEEListener	0	Does the board support the ListenerPresent function? If not, ListenerPresent will always return true.
IEEE488SD	1	Does the board support the 488SD high-speed protocol?
IEEEDMA	2	Does the board use DMA hardware?
IEEEFIFO	3	Does the hardware have a FIFO buffer to support higher speed transfers?
IEEEIOBASE	100	Return the base I/O address
IEEETIMEOUT	200	Return the current timeout setting
IEEEINPUTEOS	201	Return the current input EOS
IEEEOUTPUTEOS1	202	Return the current output EOS1
IEEEOUTPUTEOS2	203	Return the current output EOS2
IEEEBOARDSELECT	204	Return the current board number
IEEEDMACHANNEL	205	Return the current DMA channel

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

see examples in earlier section on ListenerPresent

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

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

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

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

The program TRTEST, installed as part of the CEC488 software, is a good program to experiment with if you want to test the use of **Send** and **Enter** with your instruments.

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

TRANSMIT (command,status)

where:

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

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

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

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

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

Visual BASIC:

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

Delphi:

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

C or C++:

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

LISTEN

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

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

TALK

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

Examples:
"TALK 3"
"TALK 9"

SEC

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

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

UNT

Untalk. Disables the current talker, if any.

UNL

Unlisten. Disables any currently assigned listeners.

MTA

My Talk Address. Assigns your board as the talker.

MLA

My Listen Address. Assigns your board as a listener.

DATA

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

END

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

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

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

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

"DATA 'testing' 13 10"

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

"DATA 27 '&k2S'"

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

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

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

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

Visual BASIC

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

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

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

C or C++

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

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

REN

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

EOI

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

GTL

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

Examples

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

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

"DATA 'hello' EOI 10"

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

"UNL LISTEN 5 8 GTL"

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

SPE

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

SPD

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

Example

"UNL MLA TALK 5 SPE"

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

PPC

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

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

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

PPD

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

PPU

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

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

Examples

"PPU"

This command disables all parallel poll responses.

"UNL LISTEN 1 2 PPC PPD"

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

"UNL LISTEN 5 PPC CMD 105"

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

DCL

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

LLO

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

CMD

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

GET

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

SDC

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

TCT

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

IFC

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

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

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

"DCL"

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

"UNL LISTEN 6 SDC"

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

"UNL LISTEN 9 GET"

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

RECEIVE (recv,maxlength,length,status)

where:

- recv is a string variable which will contain the received data. recv must be initialized to a string containing at least as many characters as you want 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 want to receive. 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

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 nonzero, and status will be 8. To continue receiving simply call **Receive** again to get the rest of the data.

Visual BASIC:

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

Delphi:

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

C or C++:

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

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

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

Tarray

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

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

where:

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

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

Visual BASIC:

```
DIM INFO%(1000)       ' array containing data
... fill the array with data ...
CALL TRANSMIT ("MTA LISTEN 8",status%)
CALL TARRAY (INFO%(1),2000,1,status%)
```

Delphi:

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

C or C++:

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

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

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

where:

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

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.

Visual BASIC:

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

Delphi:

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

C or C++:

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

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

Sometimes, the data sent by the instrument is not in a format which can readily be used by the PC or by the language you are writing in. 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.

In these cases, you will have to reformat the data, accessing it as individual bytes and rearranging them so the computer can process them.

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

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

To get the best data rates, use the **Tarray** and **Rarray** routines. The other routines (Send and Enter) are designed for ease of use and mainly for shorter text strings. Tarray and Rarray are optimized to use the high speed hardware features of each board. Transfer speed is usually limited by the GPIB device. Most instruments have maximum rates of 100K bytes/second or less. All our boards can transfer data many times faster than this rate.

16-bit ISA bus interface (488EX)

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

PCI bus interface

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

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

DMACHANNEL (channel)

where:

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

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

Visual BASIC:

```
CALL DMACHANNEL(1)
```

Delphi:

```
dmachannel(1);
```

C or C++:

```
dmachannel(1);
```

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

SetPort

SETPORT (board,port)

where:

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

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

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

Note: Even if you do not call SetPort() in your program, your application can still be used on boards set to nondefault I/O addresses. See the following section on configuration files.

Visual BASIC:

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

Delphi:

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

C or C++:

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

BOARDSELECT (board)

where:

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

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

Visual BASIC:

```
CALL BOARDSELECT (2)
```

Delphi:

```
boardselect (2);
```

C or C++:

```
boardselect (2);
```

SETTIMEOUT (msec)

where:

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

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

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

Visual BASIC:

```
CALL SETTIMEOUT (3000)
```

Delphi:

```
settimeout (3000);
```

C or C++:

```
settimeout (3000);
```

SETOUTPUTEOS (eos1,eos2)

where:

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

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

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

Visual BASIC:

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

Delphi:

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

C or 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: Enter** and **Receive** also terminate reception when they receive a pre-defined number of characters, or when the GPIB EOI signal is received.

Visual BASIC:

```
CALL SETINPUTEOS (13)
```

Delphi:

```
setinputeos (13);
```

C or C++:

```
setinputeos (13);
```

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

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

The process of finding boards works as follows:

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

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

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

DOS

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

Windows

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

Windows NT/2000/XP

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

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

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

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

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

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

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

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

The tables on the following page show the internal registers of the interface chip on the card. These tables will be referred to during the remainder of this discussion. Only those bits which are important to the discussion in this section are highlighted.

The GPIB interface chip is accessed with output and input instructions. In DOS and early versions of Windows, I/O instructions were directly available in the programming language. For example:

DOS BASIC:

```
OUT &H2BB,&H40        ' set spoll response
```

However, in modern languages, you cannot directly access I/O ports. We provide subroutines you can call to access the board registers, however.

Microsoft Visual C:

```
extern declspec(dllimport)
      unsigned char ieee488_inp( short offset );
extern declspec(dllimport)
      void ieee488_outp( short offset,
                  unsigned char value );
  . . .
    ieee488_outp( 3, 0x40 );   // set spoll response
```

Input:

Offset	Name	Bit assignment							
0	data	D7	D6	D5	D4	D3	D2	D1	D0
1	status 1	---	---	GET	END	DEC	ERR	DO	DI
2	status 2	---	SRQ	---	---	CO	---	---	ADC
3	spoll status	S8	PEND	S6	S5	S4	S3	S2	S1
4	address stat	CIC	---	---	---	---	LA	TA	---
5	command pass	---	---	---	---	---	---	---	---
6	address 0	---	---	---	---	---	---	---	---
7	address 1	---	---	---	---	---	---	---	---

Output:

0	data	D7	D6	D5	D4	D3	D2	D1	D0
1	mask 1	---	---	GET	END	DEC	---	DO	DI
2	mask 2	---	SRQ	---	---	CO	---	---	ADC
3	spoll resp.	S8	RSV	S6	S5	S4	S3	S2	S1
4	address mode	---	---	---	---	---	---	---	---
5	aux. command	D7	D6	D5	D4	D3	D2	D1	D0
6	address 0/1	---	---	---	---	---	---	---	---
7	end of string	---	---	---	---	---	---	---	---

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

Microsoft Visual C:

```
while (1) {
      if (ieee488_inp(4) & 2)
            ... /* talk */
      if (ieee488_inp(4) & 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.

However, this method assumes that you always receive data followed by sending data, in strict sequence, so that you always have plenty of time to recognize and respond to being addressed.

If you want 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 and DI bits on the interface chip indicate readiness to send or receive data bytes.

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 interface 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.

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 example which follows shows a device which can receive or transmit in any sequence.

Microsoft Visual C:

```
#include "ieee-c.h"
extern declspec(dllimport)
 unsigned char ieee488_inp( short offset );
extern declspec(dllimport)
 void ieee488_outp(short offset,unsigned char value );
#define MYADDR 16
int status;
char inbuf[256],outbuf[256],lastbyte;
main () {
   initialize (MYADDR,2);
   status = inbuf[0] = outbuf[0] = 0;
   while (1) {
      talk();
      listen();
      if (inbuf[0]) process();   }
}
talk() {
      status |= ieee488_inp(1);
      if (status & 4) {   // ERR - retransmit
         memmove (outbuf+1,outbuf,255);
         outbuf[0] = lastbyte;
         status &= 0xF7;  }
      if (!outbuf[0]) return(0);
      if (!(status & 2)) return(0);  // DO?
      status &= 0xFD;
      ieee488_outp (0,outbuf[0]);
      lastbyte = outbuf[0];
      strcpy (outbuf,outbuf+1);
}
listen() {
      if (strlen(inbuf)==255) return(0);
      status |= ieee488_inp(1);
      if (!(status & 1)) return(0);  // DI?
      status &= 0xFE;
      inbuf[strlen(inbuf)+1] = '\0';
      inbuf[strlen(inbuf)] = ieee488_inp(0);
}
process() {

   // note: assumes ASCII strings with LF terminator
   char *p,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");
}
```

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 interface chip. 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 process() routine.

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.

talk() routine: Talking. If the ERR bit is set, the last byte is put back in the output buffer. 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.

listen() routine: Listening. Similar to talking, above.

process() routine: 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 process() routine as needed for your application.

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.

```
ieee488_outp( 3, 0x41);   // set spoll response and
                  // request service
```

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

```
// wait until polled ---
while( ieee488_inp(3) & 0x40 ) ;
```

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: 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.

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.

```
transmit( "TALK 4 TCT", &status );
```

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

Interrupt Processing

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

However, handling interrupts in Windows is a very advanced programming concept, and generally requires writing device-driver level code. In the days of DOS, you could handle interrupts directly in user programs, but no longer. This manual does not cover advanced Windows programming concepts. There are many good books on the subject in any technical bookstore.

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

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

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

For example, in Visual BASIC:

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

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

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

After wisdom comes wit.
- Evan Esar

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

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

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

```
'-----------------------------------------------
' Example 1: use of SEND & ENTER to communicate
' with an instrument (Keithley 195 meter)
'-----------------------------------------------
CALL initialize( 21, 0 ) ' initialize as controller
CALL send( 16, "F0R0X", status% )
IF status%<>0 THEN STOP     ' test for errors
CALL enter( r$, l%, 16, status% )
IF status%<>0 THEN STOP
PRINT "Data received='";r$;"'"
```

```
'---------------------------------
' Example 2: testing for a service request (SRQ)
'---------------------------------
CALL initialize( 21, 0 )
CALL send( 4, "measure", status% )
IF status%<>0 THEN STOP
'--- now, wait for SRQ status bit ---
'
while( not srq() )
   DoEvents
wend
'
' -- SRQ has occurred, now read the result
'
CALL enter(R$,L%,4,status%)
IF status%<>0 THEN STOP
PRINT "Data received='";R$;"'"
```

```
'-----------------------
' Example 3: acquiring data for use with another
' program, such as Excel.
'-----------------------
CALL initialize(21,0)
'
OPEN "DATAFILE.PRN" FOR OUTPUT AS #1 ' open a file
'
'---- Loop to acquire 100 values ---
FOR I=1 TO 100
  CALL enter(R$,L%,16,status%)
  IF status%<>0 THEN STOP
  PRINT #1,R$
NEXT I
CLOSE
'
' The data is now in the disk file.
```

```
'-----------------------------------------------
' Example 4: use of TRANSMIT
'-----------------------------------------------
CALL initialize( 21, 0 )
'
' Now, send a trigger to three devices
' at 2,4, and 5, using the LISTEN and GET (group
' execute trigger) commands.
'
CALL transmit("REN UNL LISTEN 2 4 5 GET",status%)
```

```
'-------------------------------------------
' Example 5: receiving binary array data
' (from a Tektronix 7612D digitizer)
'-------------------------------------------
 CALL initialize( 21, 0 )
'
' device clear
 CALL TRANSMIT("REN LISTEN 1 SEC 0 SDC",status%)
'
' Start digitizing
 CALL TRANSMIT("MTA DATA 'REP 0,A' END",status%)
'
 DIM DATA%(1024)              ' array for data
' Set device to talk
 CALL TRANSMIT("MLA TALK 1 SEC 0",status%)
' read binary data
 CALL RARRAY(DATA%(),2048,L%,status%)
' Note: the data is sent in a non-PC binary format
```

```
PROGRAM example1;
USES ieeedel;
{
      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 ieeedel;
{
      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 ieeedel;
{
  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 and can be read by
spreadsheets or similar programs.
}
END.
```

```
PROGRAM example4;
USES ieeedel;
{
      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 ieeedel;
{
      Example 5: receiving binary array data
      (from a Tek 7612D digitizer)
}
CONST
      sdc = 'REN LISTEN 1 SEC 0 SDC';
VAR
      status : integer;
      l : word;
      data : array [1..1024] of integer;
BEGIN
  initialize (21,0);
  transmit (sdc,status);   { selected device clear }

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

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

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

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

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

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

#define K195 16

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

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

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

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

  initialize (21,0);
  send (4,"MEASURE",&status); /* start measurement */
  while (!(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;i<100;i++)
      {
         enter (r,80,&l,16,&status);     /* read */
         fprintf (datafile,"%s\n",r);    /* store */
      }
      fclose (datafile);
/*
 The data is now in the disk file.
*/
}
```

```
/*
      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);
}
```

Note: these examples are shown as code fragments which may be placed inside any desired C# routine, such as the Click procedure for a button.

```
// Example 1: use of SEND & ENTER to communicate
// with an instrument (Keithley 195 meter)

  int status,l;
  String r;

  CEC488.initialize (21,0);   // make PC controller
  CEC488.send (16,"F0R0X", out status);
                        // device command
  CEC488.enter ( out r,80, out l,16, out status);
                        // read a voltage
```

```
//    Example 2: testing for a service request (SRQ)

  int status,l;
  String r;

  CEC488.initialize (21,0);
  CEC488.send (4,"MEASURE", out status);
                    // start measurement
  while (!CEC488.srq()) ;     // wait for SRQ
  CEC488.enter (out r,80, out l,4, out status);
                    // read results
```

```
//    Example 3: use of TRANSMIT

  int status;
  CEC488.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.

  CEC488.transmit ("REN UNL LISTEN 2 4 5 GET",
            out status);
```

```
//    Example 4: receiving binary array data
//          (from a Tek 7612D digitizer)


 String sdc = "REN LISTEN 1 SEC 0 SDC";
 byte data[];
 int status,l;

  CEC488.initialize (21,0);
  CEC488.transmit (sdc, out status);  // device clear

  // start digitizing
  CEC488.transmit ("MTA DATA 'REP 0,A' END",
                out status);
  // set device to talk
  CEC488.transmit ("MLA TALK 1 SEC 0", out status);
  // read 3 bytes of header info
  CEC488.rarray (out data,3, out l, out status);
  // read 2048 bytes of waveform data
  CEC488.rarray (out data,2048, out l, out status);
  // clear device to stop it
  CEC488.transmit (sdc, out status);
```

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

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

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

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 is less than or equal to 1 Mbit/second.

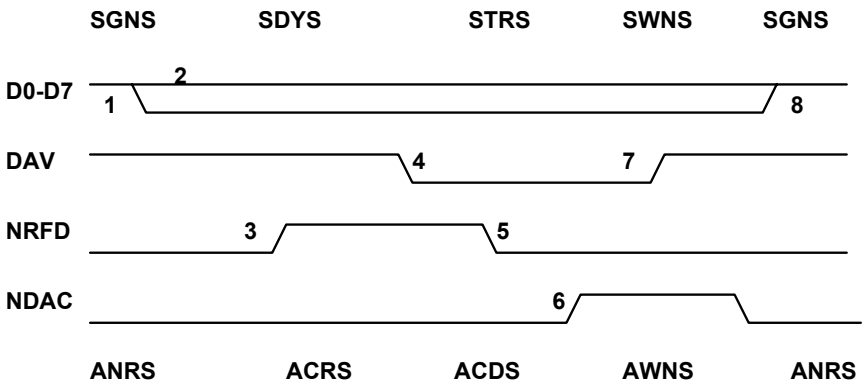
Although the interface was originally designed for instruments, it has become a well received standard for computer peripheral communication. A computer, acting as the active controller of the bus, can logically address up to 31 primary devices each with up to 31 secondary addresses. The peripherals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

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

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

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

Handshake Timing Sequence



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

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

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

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

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

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

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

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

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

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

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

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

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

Data Bus

DIO1 through DIO8 - bidirectional asynchronous data lines.

Management Bus

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

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

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

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

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

Transfer Bus

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

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

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

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

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

Interface files and setup

Visual BASIC

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

VB\IEEEVB.BAS	**Add this file to your project using the File / Add File menu command**

Note: If you are using VB 3.0, use the file IEEEVB3.BAS instead. If you are using VB.NET, use the file IEEEVB.VB instead.

Other BASIC versions

All older DOS or 16-bit Windows BASIC versions require CEC488 v7.0, which can be downloaded from the web at www.cec488.com, along with documentation.

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 want. For example, you can call INITIALIZE as either:

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

or

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

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

INITIALIZE

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

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

SEND

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

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

ENTER

For Visual BASIC:

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

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

SPOLL

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

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

PPOLL

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

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

TRANSMIT

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

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

RECEIVE

For Visual BASIC:

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

- "r$" is the string into which the received data will be placed.

- "maxlen%" (VALUE) is the maximum number of characters to receive.
- "l%" is a variable which will be set to the actual received length.
- "status%" indicates the success or failure of the data transfer.

TARRAY

```
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

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

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

SRQ%

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

or

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

SETPORT

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

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

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

BOARDSELECT

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

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

DMACHANNEL

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

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

SETTIMEOUT

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

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

SETOUTPUTEOS

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

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

SETINPUTEOS

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

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

LISTENERPRESENT%

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

- "addr%" is the GPIB address to be tested.

- "present%" is the return value, which indicates whether a listener was present at that address.

GPIB BOARD PRESENT%

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

ENABLE 488EX

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

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

Examples - Visual BASIC

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

```
CALL INITIALIZE (21,0)
CALL SEND (16,"F0R0X",status%)
CALL ENTER (r$,80,1%,16,status%)
PRINT "Received data is '";r$;"'"
```

Using TRANSMIT and RARRAY to receive binary data:

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

Interface files

Borland Delphi (32-bit) for Windows

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

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

Now put the following line in your program:

```
uses ieeedel;
```

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

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

or

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

Note: Integer constants do not contain a decimal point.

INITIALIZE

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

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

SEND

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

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

ENTER

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

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

SPOLL

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

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

PPOLL

```
ppoll (poll);
```

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

TRANSMIT

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

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

RECEIVE

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

- "rstring" (string) is the string into which the received data will be placed.

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

TARRAY

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

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

RARRAY

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

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

SRQ

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

SETPORT

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

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

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

BOARDSELECT

```
boardselect (board);
```

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

DMACHANNEL

```
dmachannel (ch);
```

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

SETTIMEOUT

```
settimeout (msec);
```

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

SETOUTPUTEOS

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

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

SETINPUTEOS

```
setinputEOS (eos);
```

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

LISTENER PRESENT

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

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

GPIB BOARD PRESENT

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

ENABLE 488EX

```
enable_488ex (e);
```

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

GPIBFeature

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

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

Examples

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

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

Using TRANSMIT and RARRAY to receive binary data:

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

C and C++ Language Interface

This section describes the use of C with our IEEE 488 interfaces.

Interface files

The C support files are installed as part of the normal SETUP. You should copy the necessary files to the working directory you will be using to develop your program.

You will need:

C\IEEE-C.H	**Include this file in your source code: #include "ieee-c.h"**
IEEE_32M.LIB	Link this when using Microsoft Visual C++ for **Win32** programs.
IEEE_32B.LIB	Link this when using Borland C++ for **Win32** programs.

Note: DOS and Win-16 support are available on CEC488 v7.0 and earlier only. You can download v7 at www.cec488.com.

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

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

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

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

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

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

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

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

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

or

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

Note: Integer constants do not contain a decimal point.

INITIALIZE

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

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

SEND

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

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

ENTER

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

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

SPOLL

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

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

PPOLL

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

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

TRANSMIT

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

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

RECEIVE

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

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

TARRAY

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

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

RARRAY

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

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

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

SRQ

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

SETPORT

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

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

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

BOARDSELECT

```
boardselect (board);
```

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

DMACHANNEL

```
dmachannel (ch);
```

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

SETTIMEOUT

```
settimeout (msec);
```

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

SETOUTPUTEOS

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

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

SETINPUTEOS

```
setinputEOS (eos);
```

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

LISTENER PRESENT

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

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

GPIB BOARD PRESENT

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

ENABLE 488EX

```
enable_488ex (e);
```

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

GPIBFeature

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

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

Examples

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

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

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

Using TRANSMIT and RARRAY to receive binary data:

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

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

C# Language Interface

This section describes the use of Microsoft C# with our IEEE 488 interfaces.

Interface files

The C# support files are installed as part of the normal SETUP. You should copy the necessary files to the working directory you will be using to develop your program.

You will need:

CEC488.cs	**Add this file to your project**

The following code shows the calling sequence for each IEEE 488 interface subroutine. Those arguments which are “out” parameters may be passed as constants rather than variables if you want. For example, you can call SEND as either:

```
CEC488.send (addr,str, out status);
```

or

```
CEC488.send (16,"this is a test", out status);
```

INITIALIZE

```
CEC488.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

```
CEC488.send (addr,info, out status);
```

- "addr" (int) is an integer indicating the IEEE 488 address of the device to send the data to.
- "info" 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

```
CEC488.enter (out rstring,maxlen, out l,addr,
     out status);
```

- "rstring" is the String into which the received data will be placed.
- "maxlen" (int) is the maximum number of characters desired.
- "l" (int) 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

```
CEC488.spoll (addr,out poll, out status);
```

- "addr" (int) 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" (int) is a variable which indicates the success or failure of the data transfer.

PPOLL

```
CEC488.ppoll (out poll);
```

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

TRANSMIT

```
CEC488.transmit (cmdstring,out status);
```

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

RECEIVE

```
CEC488.receive (out rstring,maxlen,out l, out status);
```

- "rstring" is the string into which the received data will be placed.
- "maxlen" (int) is the maximum number of characters desired.
- "l" (int) 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

```
CEC488.tarray (d,count,eoi,out status);
```

- "d" (byte []) is a byte array variable containing the data to be transmitted.
- "count" (int) is the number of bytes to be transmitted.
- "eoi" (int) 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

```
CEC488.rarray (out d,count,out l,out status);
```

- "d" (byte []) is the byte array variable into which data will be received.
- "count" (int) is the maximum number of bytes to be received.
- "l" (int) 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 (CEC488.srq()) ... { put any statement here }
```

SETPORT

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

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

```
CEC488.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" (int) is the I/O port address of the IEEE 488 board.

BOARDSELECT

```
CEC488.boardselect (board);
```

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

DMACHANNEL

```
CEC488.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

```
CEC488.settimeout (msec);
```

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

SETOUTPUTEOS

```
CEC488.setoutputEOS (eos1,eos2);
```

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

SETINPUTEOS

```
CEC488.setinputEOS (eos);
```

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

LISTENER PRESENT

```
present = CEC488.listener_present (addr);
```

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

GPIB BOARD PRESENT

```
if (!CEC488.gpib_board_present()) ...
```

GPIBFeature

```
value = CEC488.gpib_feature (feature);
```

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

Examples

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

```
int status;
int l;
String r;

CEC488.initialize (21,0);
CEC488.send (16,"F0R0X", out status);
CEC488.enter (out r,79, out l,16, out status);
```

Using TRANSMIT and RARRAY to receive binary data:

```
int status;
int l;
byte d[];

CEC488.initialize (21,0);
// send a command to the device
CEC488.transmit (
  "REN MTA LISTEN 3 DATA 'READ' END",
   out status);
// set device to talk and read the data
CEC488.transmit ("MLA TALK 3", out status);
CEC488.rarray (out d,2000, out l, out status);
```

Troubleshooting

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

Checklist for Solving 488 Programming Problems

- ✓ Try running TEST488. If it reports a board problem, check the switch settings and check for conflicts with another add-in card.
- ✓ Swap components if possible. If you have two 488 cards, two cables, and/or two instruments, try swapping them to see if any one component is faulty.
- ✓ If some device commands work, but not others, it is **NOT** a 488 board problem. It is almost certainly something wrong with the command or data format, possibly the wrong terminating character(s).
- ✓ If a SEND call returns status 0, but nothing happens on the device, THEN the device has accepted all the characters - therefore, the problem is in the format of the command. Either the command is incorrect (a typo), or the device is still waiting for a terminator. Try adding a return character to the string, or possibly a semicolon.
- ✓ If the error happens repeatably, but not every time, it is probably also a format error. For example, if ENTER returns the right data once, then returns a null string, then the right data, and so on... this means that the first ENTER call did not read all the characters sent by the device. Changing the input terminator might help.
- ✓ If the problem looks like a question of programming the correct device command sequence, look for coding examples in the device's manual. Even if the examples are written for a computer like the HP-85, they may be useful.

☞ **Computer fails to boot correctly.**

There is a conflict between the 488 board and other hardware in your PC. **This should NOT happen with plug-and-play boards like the PCI bus interface card.** It may be:

- A memory conflict, usually with a display adapter, network adapter, or expanded memory board. Try a different memory address setting, such as location D000 or C800. You can also disable the ROM memory on 488EX (16-bit ISA bus card). PCI488 and PS488 cannot have a memory conflict.
- A DMA conflict, usually with a network adapter or data acquisition adapter. Try disabling DMA. On PC488 and 488EX this is done by removing jumpers. PCI488 and PS488 cannot have a DMA conflict. If disabling DMA works, you can try using a different DMA channel.
- An I/O conflict. This is very unusual. If there is an I/O conflict, choose another I/O address, such as 2A8, and see the section on CEC488.INI files in chapter 3.

☞ **The GPIB device does not respond.**
(Check the status returned by the IEEE 488 subroutine)

This can be an incorrect device address

- Check the device address switches. You may also want to try the board with a different GPIB device if you have one available.

Or a programming problem

- Try the same SEND and/or ENTER operations with the TRTEST program provided with the board. If these work, double-check your code and look again at the example programs in the language interface sections (A-F) of the manual.

Or a device problem

- Try the board with a different GPIB device if you have one available.

Or a cable problem

- Swap cables if you have another cable available. Inspect the cable and the connectors on the computer and the device for bent pins.

Or a board problem

- Check the board switch settings. Check that the board is firmly seated in the computer's add-in slot. You may want to try the board in another slot in the computer. Also, try cleaning the edge connector of the board with a regular pencil eraser to make sure a good electrical contact is

obtained. Run the TEST488 program to see if the board is OK. If TEST488 is OK, there could still possibly be a problem in the final output stages of the board. Make sure you have tried the other possibilities listed above before resorting to factory repair.

☞**"Unresolved external: IEEE488_..." when linking.**

You must link the IEEE488.LIB file (for DOS), or WIN488.LIB (for 16-bit Windows), or IEEE_32M.LIB (for 32-bit Windows), which contains these subroutines.

☞**"Unresolved external: initialize..." in C.**

You MUST include the file IEEE-C.H in your source code. Do NOT edit this file.

☞ **488 ROM not found, or not accessible. Note: The ROM is not needed for most programming languages. However, it may be used with some very old programs, and some languages, such as Turbo BASIC.**

This can be caused by a hardware memory conflict. Try a different memory address setting, such as location D000 or C800.

On 488EX, make sure the ROM is not disabled (switch S1-7 should be ON to enable the ROM).

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

If you are running a 386 memory manager such as QEMM-386 or 386Max, it may be hiding the 488 ROM. These memory manager software packages have command-line parameters that allow you to exclude regions of memory. For example, with QEMM, use "X=CC00-CDFF" to exclude the CC00 segment of memory.

This manual covers only the plug-and-play models of GPIB hardware, such as PCI boards, USB modules, etc. These models have no user-configurable settings or switches. Older hardware, such as ISA plug-in boards, which do have hardware address switches, are covered in the manual for CEC488 v7.0, which is shipped with those cards and available for download from the web at www.cec488.com.

Forcing a specific I/O address

PCI488 is plug-and-play, so it will configure at whatever I/O address is chosen by the computer at power-up. If you have a very old program written for our 488 cards **and** you do not have source code (so you cannot simply recompile the program), you may want to force the card to configure at a known I/O address, such as 2B8 hex (the default setting for our older cards). You can do this with the PCICONF.EXE utility program:

```
C:\Program Files\CEC488>  pciconf  2B8
```

ASCII	Hex	Decimal	GPIB
NULL	00	0	
SOH	01	1	GTL
STX	02	2	
ETX	03	3	
EOT	04	4	SDC
ENQ	05	5	PPC
ACK	06	6	
BELL	07	7	
BS	08	8	GET
HT	09	9	TCT
LF	0A	10	
VT	0B	11	
FF	0C	12	
CR	0D	13	
SO	0E	14	
SI	0F	15	
DLE	10	16	
DC1	11	17	LLO
DC2	12	18	
DC3	13	19	
DC4	14	20	DCL
NAK	15	21	PPU
SYNC	16	22	
ETB	17	23	
CAN	18	24	SPE
EM	19	25	SPD
SUB	1A	26	
ESC	1B	27	
FS	1C	28	
GS	1D	29	
RS	1E	30	
US	1F	31	

ASCII	Hex	Decimal	GPIB
space	20	32	LA0
!	21	33	LA1
"	22	34	LA2
#	23	35	LA3
$	24	36	LA4
%	25	37	LA5
&	26	38	LA6
'	27	39	LA7
(	28	40	LA8
)	29	41	LA9
*	2A	42	LA10
+	2B	43	LA11
,	2C	44	LA12
-	2D	45	LA13
.	2E	46	LA14
/	2F	47	LA15
0	30	48	LA16
1	31	49	LA17
2	32	50	LA18
3	33	51	LA19
4	34	52	LA20
5	35	53	LA21
6	36	54	LA22
7	37	55	LA23
8	38	56	LA24
9	39	57	LA25
:	3A	58	LA26
;	3B	59	LA27
<	3C	60	LA28
=	3D	61	LA29
>	3E	62	LA30
?	3F	63	UNL

ASCII	Hex	Decimal	GPIB
@	40	64	TA0
A	41	65	TA1
B	42	66	TA2
C	43	67	TA3
D	44	68	TA4
E	45	69	TA5
F	46	70	TA6
G	47	71	TA7
H	48	72	TA8
I	49	73	TA9
J	4A	74	TA10
K	4B	75	TA11
L	4C	76	TA12
M	4D	77	TA13
N	4E	78	TA14
O	4F	79	TA15
P	50	80	TA16
Q	51	81	TA17
R	52	82	TA18
S	53	83	TA19
T	54	84	TA20
U	55	85	TA21
V	56	86	TA22
W	57	87	TA23
X	58	88	TA24
Y	59	89	TA25
Z	5A	90	TA26
[	5B	91	TA27
\	5C	92	TA28
]	5D	93	TA29
^	5E	94	TA30
_	5F	95	UNT

ASCII	Hex	Decimal	GPIB
`	60	96	SC0
a	61	97	SC1
b	62	98	SC2
c	63	99	SC3
d	64	100	SC4
e	65	101	SC5
f	66	102	SC6
g	67	103	SC7
h	68	104	SC8
i	69	105	SC9
j	6A	106	SC10
k	6B	107	SC11
l	6C	108	SC12
m	6D	109	SC13
n	6E	110	SC14
o	6F	111	SC15
p	70	112	SC16
q	71	113	SC17
r	72	114	SC18
s	73	115	SC19
t	74	116	SC20
u	75	117	SC21
v	76	118	SC22
w	77	119	SC23
x	78	120	SC24
y	79	121	SC25
z	7A	122	SC26
{	7B	123	SC27
|	7C	124	SC28
}	7D	125	SC29
~	7E	126	SC30
DEL	7F	127	SC31

488SD: High Speed Streaming Data Protocol

What is 488SD?

The 488SD protocol was a proposed extension to the IEEE 488 specification to allow higher speed data transfers (up to 5 MB/second). CEC made this proposal to the IEEE technical committee in 1992, and implemented the protocol on one hardware model, the 488EX board.

However, the proposal has not been widely adopted and can be considered obsolete, so no detailed coverage will be given in this manual.