Personal488 User's Manual

For Windows® 95/98/Me/NT/2000

the smart approach to instrumentation ™

IOtech, Inc.
25971 Cannon Road
Cleveland, OH 44146-1833
Phone: (440) 439-4091
Fax: (440) 439-4093
E-mail: sales@iotech.com
Internet: www.iotech.com

Personal488
User's Manual
For Windows 95/98/Me/NT/2000

p/n 495-0903 Rev. 3.0

Warranty Information

Your IOtech warranty is as stated on the *product warranty card*. You may contact IOtech by phone, fax machine, or e-mail in regard to warranty-related issues.

Phone: (440) 439-4091, fax: (440) 439-4093, e-mail: sales@iotech.com

Limitation of Liability

IOtech, Inc. cannot be held liable for any damages resulting from the use or misuse of this product.

Copyright, Trademark, and Licensing Notice

FCC Statement

IOtech devices emit radio frequency energy in levels compliant with Federal Communications Commission rules (Part 15) for Class A devices. If necessary, refer to the FCC booklet *How To Identify and Resolve Radio-TV Interference Problems* (stock # 004-000-00345-4) which is available from the U.S. Government Printing Office, Washington, D.C. 20402.

CE Notice

Many IOtech products carry the CE marker indicating they comply with the safety and emissions standards of the European Community. As applicable, we ship these products with a Declaration of Conformity stating which specifications and operating conditions apply.

Warnings, Cautions, Notes, and Tips

Refer all service to qualified personnel. This caution symbol warns of possible personal injury or equipment damage under noted conditions. Follow all safety standards of professional practice and the recommendations in this manual. Using this equipment in ways other than described in this manual can present serious safety hazards or cause equipment damage.

This warning symbol is used in this manual or on the equipment to warn of possible injury or death from electrical shock under noted conditions.

This ESD caution symbol urges proper handling of equipment or components sensitive to damage from electrostatic discharge. Proper handling guidelines include the use of grounded anti-static mats and wrist straps, ESD-protective bags and cartons, and related procedures.

This symbol indicates the message is important, but is not of a Warning or Caution category. These notes can be of great benefit to the user, and should be read.

In this manual, the book symbol always precedes the words "Reference Note." This type of note identifies the location of additional information that may prove helpful. References may be made to other chapters or other documentation.

Tips provide advice that may save time during a procedure, or help to clarify an issue. Tips may include additional reference.

Specifications and Calibration

Specifications are subject to change without notice. Significant changes will be addressed in an addendum or revision to the manual. As applicable, IOtech calibrates its hardware to published specifications. Periodic hardware calibration is not covered under the warranty and must be performed by qualified personnel as specified in this manual. Improper calibration procedures may void the warranty.

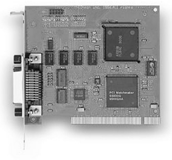
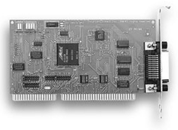
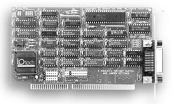
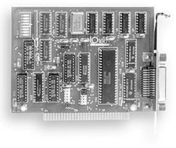
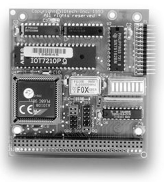
Quality Notice

IOtech has maintained ISO 9001 certification since 1996. Prior to shipment, we thoroughly test our products and review our documentation to assure the highest quality in all aspects. In a spirit of continuous improvement, IOtech welcomes your suggestions.

Your order was carefully inspected prior to shipment. When you receive your system, carefully unpack all items from the shipping carton and check for physical signs of damage that may have occurred during shipment. Promptly report any damage to the shipping agent and your sales representative. Retain all shipping materials in case the unit needs to be returned to the factory.

Table of Contents

Personal488 Overview 1

Personal488 Hardware	*Plug-and-Play*	*Bus Type and Transfer Rate*
Personal488/PCI (with PCI488) 	Yes	32-bit PCI Bus 1 Mbyte/s
Personal488/ATpnp (with AT488pnp) 	Yes	16-bit ISA Bus 1 Mbyte/s
Personal488/CARD (with CARD488)	Yes	"hot swapping" PC Card (PCMCIA) 1 Mbyte/s
Personal488/AT (with AT488) 	No	16-bit ISA-bus 1 Mbyte/s
Personal488 (with GP488B)* 	No	8-bit ISA-bus 330 Kbyte/s
Personal488/MM (with GP488B/MM)* 	No	8-bit PC/104 330 Kbyte/s

* GP488 boards with serial numbers of 036731 or lower are not supported by Drivers for Windows 9x or WindowsNT.

Note: Items pictured are not shown to the same scale.

Hardware Products

The family of Personal488 PC/IEEE 488 controller interfaces includes the six (6) interfaces which are discussed in this manual. All of them are IEEE 488.2 compatible and are supported by 32-bit Driver488 software for Windows 95, 98, Me, 2000 and NT. These interfaces are discussed in the following Personal488 packages:

Hardware Configurations

Plug-and-Play Devices	*"Non plug-and-play" Devices*
• **Personal488/PCI (PCI488)**	• **Personal488/AT (AT488)**
• **Personal488/ATpnp (AT488pnp)**	• **Personal488 (GP488B)**
• **Personal488/Card (Card488)**	• **Personal488/MM**

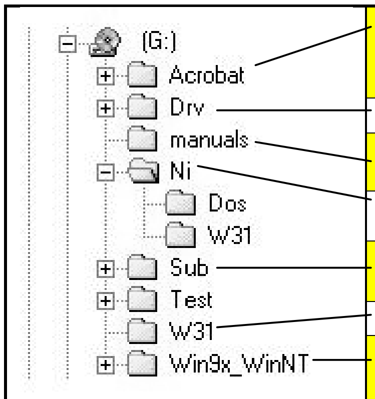
Reference Note:

- Refer to Chapter 4, *Hardware Configuration Reference* for information concerning jumpers and switches.
- Refer to Appendix D for hardware specifications.

Software Installation

The installation process consists of running an installation setup program, and for non plug-and-play products running the Add New Hardware program found in the Windows Control Panel. The installation setup program will automatically determine the version of Windows (e.g. Windows 95, 98, Me, NT, or 2000) and copy all the necessary drivers and support files to the appropriate destinations.

Install the software before installing the hardware. Since the installation setup program installs driver and INF files, plug-and-play boards will be automatically configured upon first startup, thus eliminating the need to insert the CD and browse for support files. For non plug-and-play devices all that is required to complete the installation is to run "Add New Hardware." This will notify Windows that a new device exists.



CD Structure

Acrobat: For installing Adobe Acrobat Reader. Documentation that has been included on the CD in the Adobe pdf format, can be viewed and printed with use of the Adobe Reader.

Driver488/DRV

The manuals folder typically contains a .pdf version of the Personal488 User's Manual. The Adobe Acrobat Reader is needed to print or read pdf files.

Driver488/NI

Driver488/SUB

Driver488/W31

The **Win9x_WinNT** folder includes drivers for Windows 95, 98, Me, NT, and 2000.

Note: On Windows systems with AutoPlay *enabled*, the setup program will automatically start upon insertion of the CD.

Note: The CD structure is subject to change without notice.

CD-ROM, Driver 488 Software Packages 2

IOtech, Inc. IEEE 488.2 Software Products

Personal488/PCI – 1 Mbyte/s PCI/IEEE 488.2 Board with Plug & Play, Digital I/O, & Software for PCs
Personal488/Atpnp – 1 Mbyte/s PCI/IEEE 488.2 Board with Plug & Play, Digital I/O, & Software for PCs
Personal488/AT – 1 Mbyte/s IEEE 488.2 Board & Software for PC/Ats
Personal488 – IEEE 488.2 Board & Software for PCs
Personal488/CARD – IEEE 488.2 PC-Card Interface, Cable, & Software for Notebook & Desktop PCs

This CD contains several driver software packages for DOS and Windows. The following table shows which Driver488 packages can be used with each IEEE 488 Controllers product type.

Folder (In CD Root Directory) →	**Win9x_WinNT**			**W31**	**DRV**	**SUB**	**NI** (Note1)
Supported Operating System →	**2000**	**9x & Me**	**NT**	**3x**	**DOS**	**DOS**	
Personal488 (ISA)	✓	✓	✓	✓	✓	✓	✓
Personal488AT (ISA)	✓	✓	✓	✓	✓	✓	✓
Personal488/Atpnp (ISA pnp)	✓	✓	No	No	No	No	✓
Personal488PCI (PCI)	✓	✓	✓	No	No	No	✓
Personal488CARD (PC-CARD)	✓	✓	No	✓	✓	✓	✓
Personal 488MM (PC104)	✓	✓	✓	✓	✓	✓	✓

Note 1: Information pertaining to National Instruments (NI) is provided in Appendix E.

Reference Note:
Refer to Appendix E for National Instruments information.

The CD contains all of the Driver488 packages available for current IEEE 488 Controller products. The various driver packages are organized according to the directory tree structure shown below. The location of each package is shown below, along with the name of the installation program in each directory.

Before running any of the installation programs, please look for a read-me file in the same directory as the install program. When present, it may contain important installation instructions.

Driver488 Packages

Driver488/DRV

IEEE 488.2 DOS Device Driver Software.

- Supports IOtech's AT488, GP488B, CARD488, NB488, GP488B/MM, MP488, MP488CT series boards.
- Includes "ON SRQ" program vectoring for Basic, C, Pascal.
- Compatible with all popular programming languages and spreadsheets.
- Automatically loads into high memory when available.

Driver488/SUB

IEEE 488.2 DOS Subroutine Driver.

- Supports IOtech's AT488, GP488B, CARD488, NB488, GP488B/MM, MP488, MP488CT series boards.
- Includes "ON SRQ" program vectoring for Basic, C, Pascal.
- Compatible with popular programming languages and spreadsheets such as C, Pascal and QuickBasic.

Driver488/W31

IEEE 488.2 Microsoft Windows Dynamic Link Library

- Supports IOtech's AT488, GP488B, CARD488, NB488, GP488B/MM, MP488, MP488CT series boards.
- Offers HP-style commands for high & low-level control.
- Designed for Windows' message passing, multi-tasking architecture.
- Includes language interfaces for Microsoft C, Visual C++, Visual Basic, Turbo C, Borland C++.
- Includes an interactive control application for exercising instruments and generating code.
- On-Line Help provides complete command reference as well as examples.

Win9x_WinNT Folder

This folder includes IEEE 488.2 drivers for the following operating systems: Windows 9x, Windows Me, Windows 2000, and Windows NT.

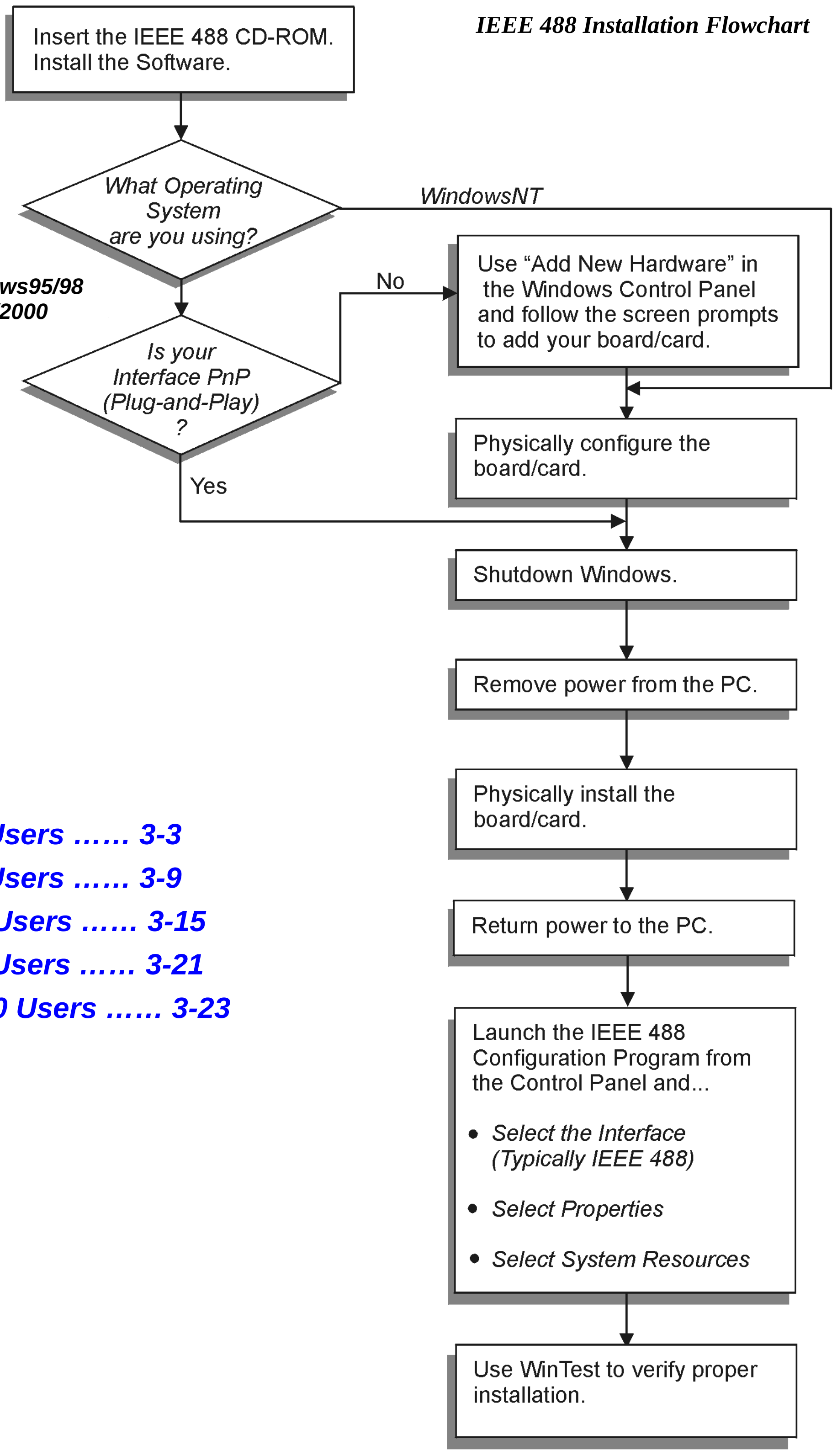
The setup program automatically detects the operating system and installs the correct drivers.

In regard to Windows 9x, Windows Me, and Windows 2000, the drivers

- Support IOtech's AT488, GP488B, GP488B/MM, PCI488 series boards.
- Integrate IEEE 488.2 control into Microsoft Windows applications.
- Provide true multi-tasking device locking.
- Were specifically designed for the 32-bit Windows environment.

Installation 3

IEEE 488 Installation Flowchart


Insert the IEEE 488 CD-ROM. Install the Software.
What Operating System are you using?
WindowsNT
Windows95/98 /Me/2000
No
Use “Add New Hardware” in the Windows Control Panel and follow the screen prompts to add your board/card.
Is your Interface PnP (Plug-and-Play) ?
Yes
Physically configure the board/card.
Shutdown Windows.
Remove power from the PC.
Physically install the board/card.
Return power to the PC.
Launch the IEEE 488 Configuration Program from the Control Panel and...
• Select the Interface (Typically IEEE 488)
• Select Properties
• Select System Resources
Use WinTest to verify proper installation.

Notes

Windows 95 Users

Software Installation

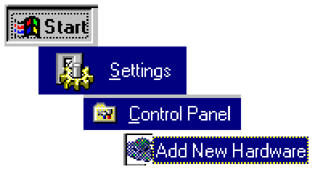
- For best results, install the interface after the software installation.
- Due to differences in Windows 95 "Add New Hardware" panels, the following description may vary slightly.
- If installing a second non plug-and-play interface, skip step 1.
- If installing a second plug-and-play interface, go to "Hardware Installation."

Step 1

Insert the IEEE488 Software CD. The CD has an auto-run program that will automatically start the setup program when the CD is inserted into the CD ROM driver. If auto-run is disabled, use Explorer to launch the Setup.exe found in the root directory of the CD. Follow the screen prompts to install the software. If non plug-and-play hardware is being installed, proceed to step 2; otherwise proceed to "Hardware Installation" on page 3-7.

Step 2

Use the "Add New Hardware" program found in the Control Panel to notify Windows 95 that you are installing new hardware. Refer to the following steps that demonstrate the typical Windows panels encountered during the "Add New Hardware" program execution:

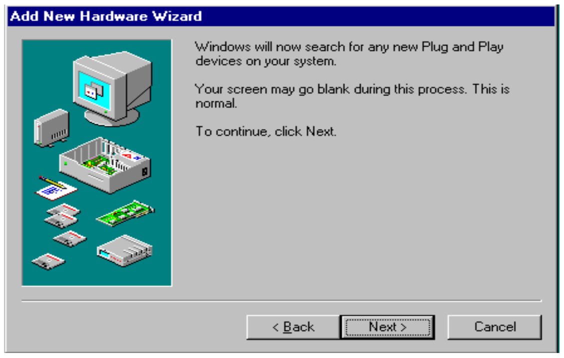


Start ⇒ Settings ⇒ Control Panel ⇒ Add New Hardware

Windows 95

Add New Hardware Procedure (*non plug-and-play users only*):

It is only necessary for users of "non plug-and-play" boards to follow the Add New Hardware Procedure. If your device is a "plug-and-play device," skip this procedure.


Add New Hardware Wizard

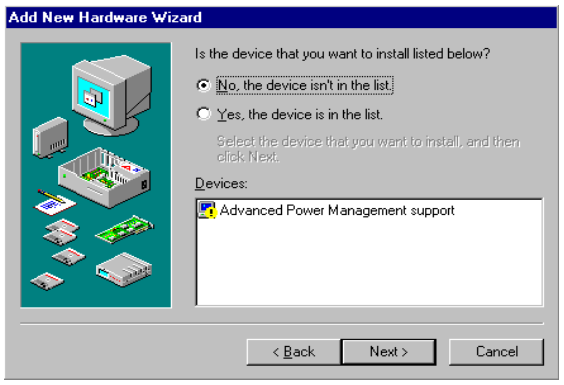
Windows will now search for any new Plug and Play devices on your system.

Your screen may go blank during this process. This is normal.

To continue, click Next.

1. The "Add New Hardware Wizard" displays an introductory message and prompts you to click ***Next***.

2. Windows 95 will automatically search for hardware. Click ***Next***.


Add New Hardware Wizard

Is the device that you want to install listed below?

[✓ No, the device isn't in the list.]

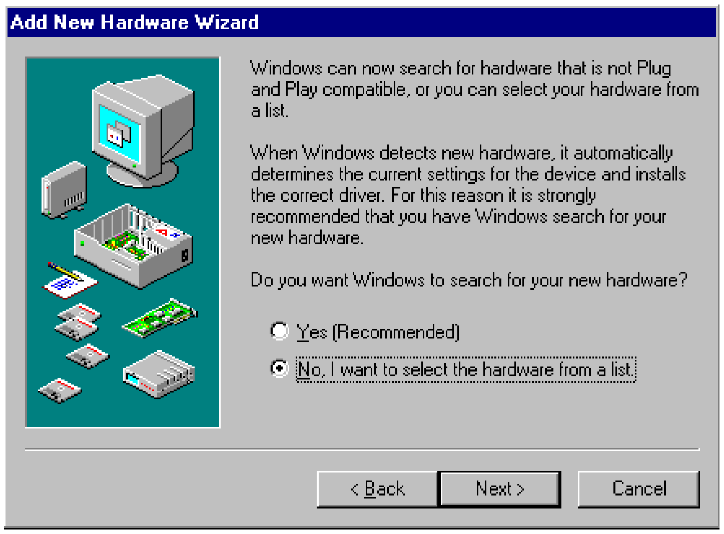
Yes, the device is in the list.

Select the device that you want to install, and then click Next.

Devices:

Advanced Power Management support

3. Click ***'No, the device isn't in the list'***


Add New Hardware Wizard

Windows can now search for hardware that is not Plug and Play compatible, or you can select your hardware from a list.

When Windows detects new hardware, it automatically determines the current settings for the device and installs the correct driver. For this reason it is strongly recommended that you have Windows search for your new hardware.

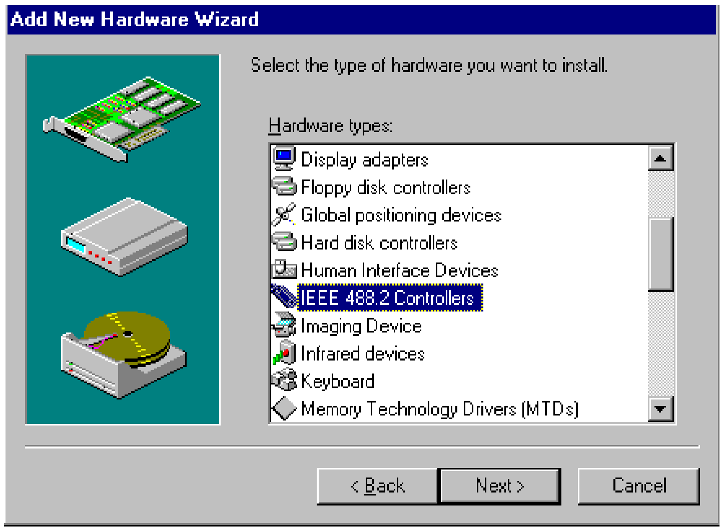
Do you want Windows to search for your new hardware?

Yes (Recommended)

[✓ No, I want to select the hardware from a list.]

4. Select the option:

   "No, I want to select the hardware from a list," then click ***Next***.



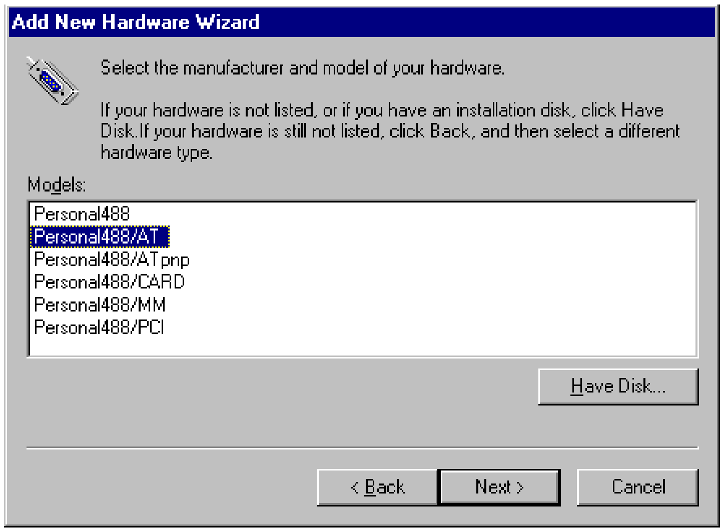
Add New Hardware Wizard

Select the type of hardware you want to install.

Hardware types:

- Display adapters
- Floppy disk controllers
- Global positioning devices
- Hard disk controllers
- Human Interface Devices
- IEEE 488.2 Controllers
- Imaging Device
- Infrared devices
- Keyboard
- Memory Technology Drivers (MTDs)

5. Choose ***IEEE488.2 Controllers*** from the list of hardware types and click ***Next.***



Add New Hardware Wizard

Select the manufacturer and model of your hardware.

If your hardware is not listed, or if you have an installation disk, click Have Disk. If your hardware is still not listed, click Back, and then select a different hardware type.

Models:

- Personal488
- Personal488/AT
- Personal488/ATpnp
- Personal488/CARD
- Personal488/MM
- Personal488/PCI

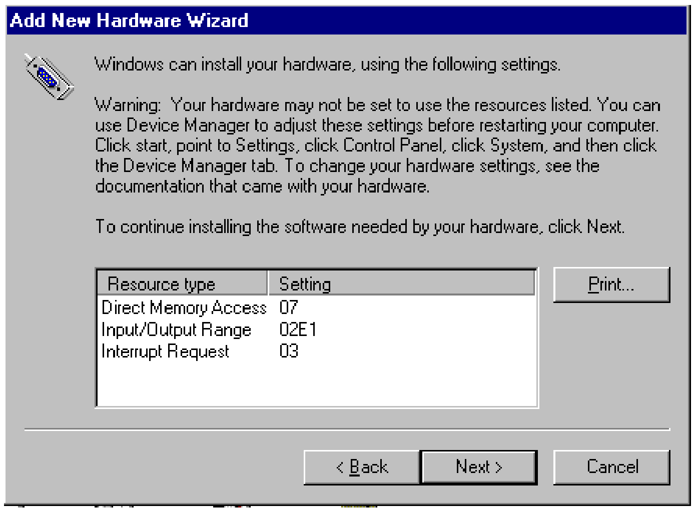
6. Windows will now display a list of devices to install.

   Select your specific Personal488 interface product.

   After making the selection, click ***Next***.

   Windows will now display the default resource settings for your interface.

Windows 95



Add New Hardware Wizard

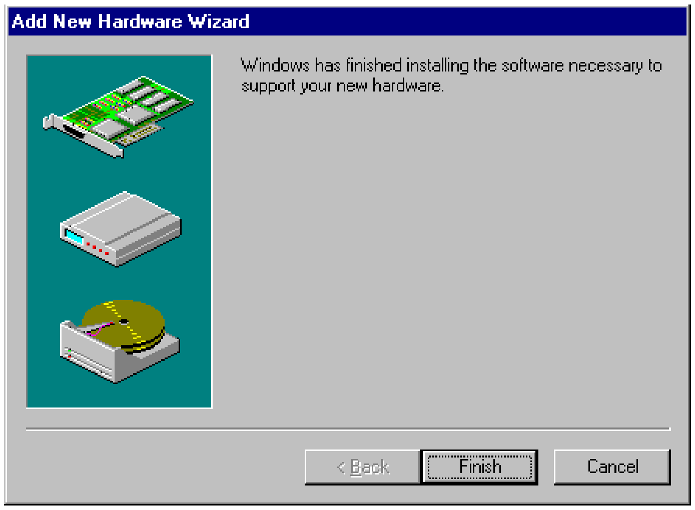
Windows can install your hardware, using the following settings.

Warning: Your hardware may not be set to use the resources listed. You can use Device Manager to adjust these settings before restarting your computer. Click start, point to Settings, click Control Panel, click System, and then click the Device Manager tab. To change your hardware settings, see the documentation that came with your hardware.

To continue installing the software needed by your hardware, click Next.

Resource type	Setting
Direct Memory Access	07
Input/Output Range	02E1
Interrupt Request	03

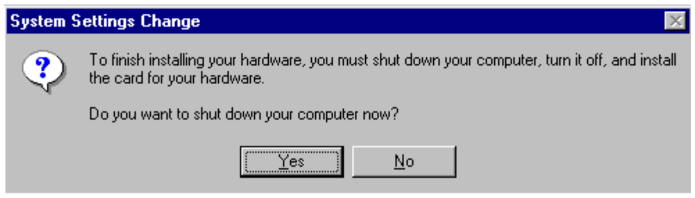
7. Make note of the displayed settings, as you must configure the jumpers and switch settings before installing an AT488 or GP488B



Add New Hardware Wizard

Windows has finished installing the software necessary to support your new hardware.

8. Click ***Finish***.



System Settings Change

To finish installing your hardware, you must shut down your computer, turn it off, and install the card for your hardware.

Do you want to shut down your computer now?

9. Click ***Yes***, proceed with Windows 95 Hardware Installation.

Hardware Installation for Windows 95 Users

Plug-and-Play Devices

Personal488/PCI
Personal488/ATpnp
Personal488/Card

1. If you have not already done so, shutdown Windows 95 after the IEEE 488 software has been successfully installed.
2. Remove power from the PC.
3. Physically install your interface. As a quick reference,

   Personal488/PCI installs into a 32-bit PCI expansion slot,
   Personal488/ATpnp installs into a 16-bit ISA expansion slot, and
   Personal488/Card installs into a PC card slot.

4. Return power to the PC. After the computer powers up, Windows will detect your new hardware.

   This completes the installation procedure.

"Non plug-and-play" Devices

Personal488/AT (AT488)
Personal488 (GP488Bplus)
Personal488/MM

1. Verify that Windows 95 has properly shutdown.
2. Remove power from the PC.
3. Physically configure the device's jumpers and switches to match the resource settings Windows reported during the driver installation.

Non plug-and-play board users: physically configure your board's jumpers and switches to match the resource settings Windows reported. If these settings conflict with other hardware, change the jumpers, switches, and Windows Resource settings to available resources.

Reference Note:
Refer to Chapter 4, *Hardware Configuration Reference* for further information concerning jumpers and switches.

4. Physically install your interface.
5. Return power to the PC.

This completes the installation procedure.

Reference Note:
See page 5-5 for instructions on running WinTest to verify proper installation.

Windows 95

Notes

Windows 98 Users

Software Installation

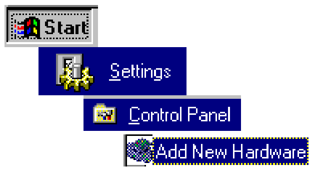
- For best results, install the interface after the software installation.
- If installing a second *non plug-and-play* interface, skip step 1.
- If installing a second *plug-and-play* interface, go to *Hardware Installation*, page 3-13.

Step 1

Insert the IEEE488 Software CD. The CD has an auto-run program that will automatically start the setup program when the CD is inserted into the CD ROM driver. If auto-run is disabled, use Explorer to launch the Setup.exe found in the root directory of the CD. Follow the screen prompts to install the software. Then, if non plug-and-play hardware is being installed, proceed to step 2; otherwise proceed to hardware installation on page 3-13.

Step 2

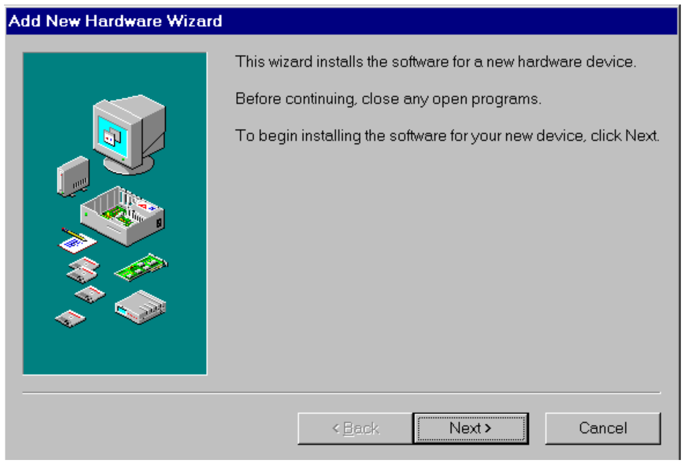
Use the "Add New Hardware" program found in the Control Panel to notify Windows 98 that you are installing new hardware. Refer to the following steps that demonstrate the typical Windows panels encountered during the "Add New Hardware" program execution:



Start ⇒ Settings ⇒ Control Panel ⇒ Add New Hardware

Add New Hardware Procedure (*non plug-and-play users only*):

It is only necessary for users of "non plug-and-play" boards to follow the Add New Hardware Procedure. If your device is a "plug-and-play device," skip this procedure.

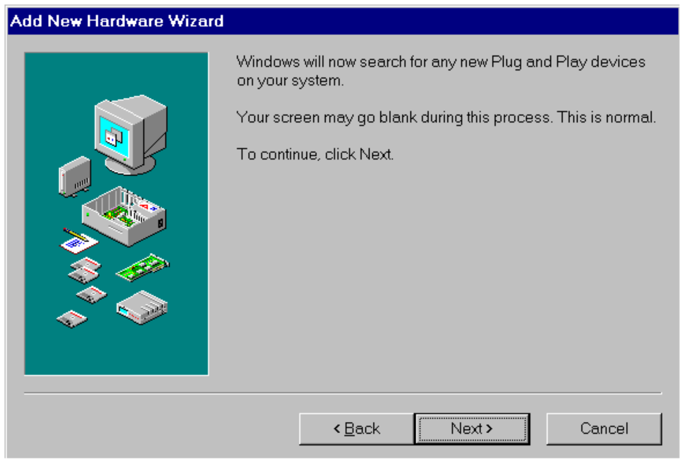


This wizard installs the software for a new hardware device.

Before continuing, close any open programs.

To begin installing the software for your new device, click Next.

1. The "Add New Hardware Wizard" displays an introductory message and prompts you to click ***Next***.



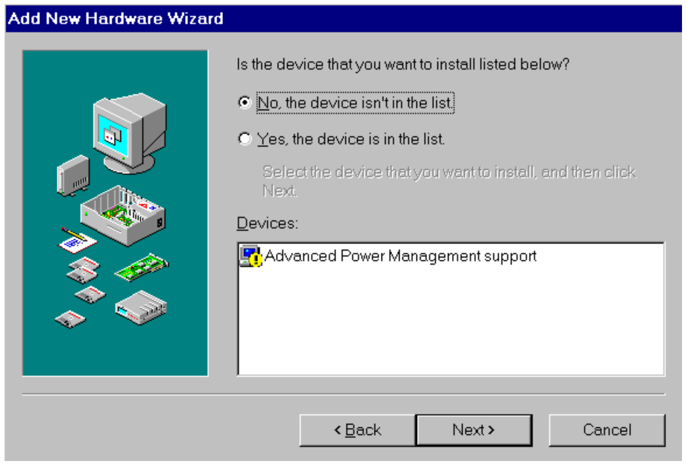
Add New Hardware Wizard

Windows will now search for any new Plug and Play devices on your system.

Your screen may go blank during this process. This is normal.

To continue, click Next.

2. Click ***Next***.



Add New Hardware Wizard

Is the device that you want to install listed below?

(●) No, the device isn't in the list.

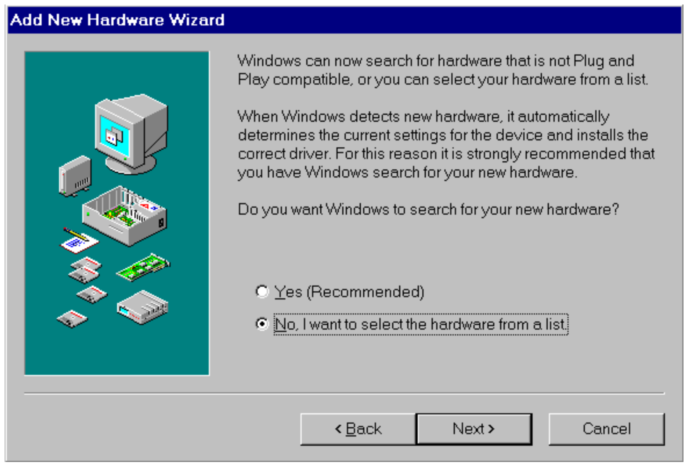
( ) Yes, the device is in the list.

Select the device that you want to install, and then click Next.

Devices:

Advanced Power Management support

3. Select '***No, the device isn't in the list***' and click ***Next***.



Add New Hardware Wizard

Windows can now search for hardware that is not Plug and Play compatible, or you can select your hardware from a list.

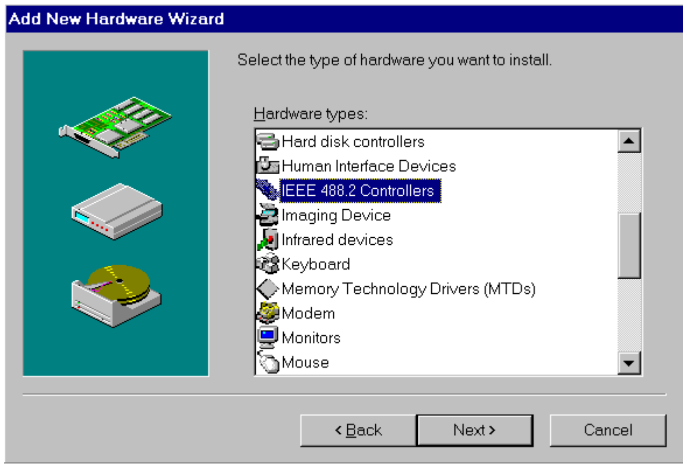
When Windows detects new hardware, it automatically determines the current settings for the device and installs the correct driver. For this reason it is strongly recommended that you have Windows search for your new hardware.

Do you want Windows to search for your new hardware?

( ) Yes (Recommended)

(●) No, I want to select the hardware from a list.

4. Select '***No, I want to select the hardware from a list'*** and click ***Next***.



Add New Hardware Wizard

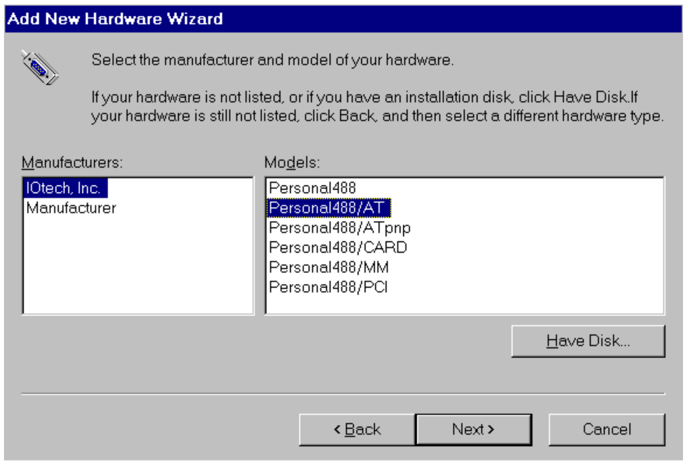
Select the type of hardware you want to install.

Hardware types:

Hard disk controllers
Human Interface Devices
IEEE 488.2 Controllers
Imaging Device
Infrared devices
Keyboard
Memory Technology Drivers (MTDs)
Modem
Monitors
Mouse

< Back | Next > | Cancel

5. Select '***IEEE 488.2 Controllers'*** and click ***Next***.



Add New Hardware Wizard

Select the manufacturer and model of your hardware.

If your hardware is not listed, or if you have an installation disk, click Have Disk. If your hardware is still not listed, click Back, and then select a different hardware type.

Manufacturers:
IOtech, Inc.
Manufacturer

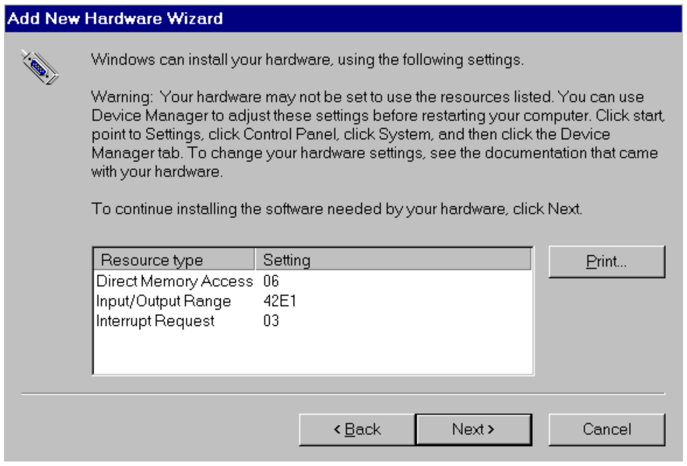
Models:
Personal488
Personal488/AT
Personal488/ATpnp
Personal488/CARD
Personal488/MM
Personal488/PCI

Have Disk...

< Back | Next > | Cancel

6. Windows will now display a list of devices to install. **Select your specific Personal488 interface product.**
7. After making the selection, click ***Next***.

Windows will now display the default resource settings for your interface.



Add New Hardware Wizard

Windows can install your hardware, using the following settings.

Warning: Your hardware may not be set to use the resources listed. You can use Device Manager to adjust these settings before restarting your computer. Click start, point to Settings, click Control Panel, click System, and then click the Device Manager tab. To change your hardware settings, see the documentation that came with your hardware.

To continue installing the software needed by your hardware, click Next.

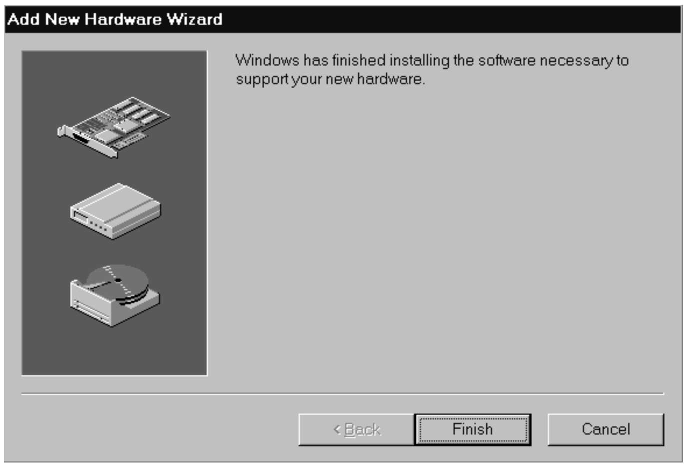
Resource type	Setting
Direct Memory Access	06
Input/Output Range	42E1
Interrupt Request	03

Print...

< Back | Next > | Cancel

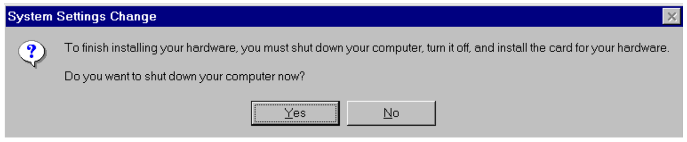
8. Make note of the displayed settings, as you must configure the jumpers and switch settings before installing an AT488 or GP488B.

Windows 98


Add New Hardware Wizard

Windows has finished installing the software necessary to support your new hardware.

9. Click ***Finish***.


System Settings Change

To finish installing your hardware, you must shut down your computer, turn it off, and install the card for your hardware.

Do you want to shut down your computer now?

10. Click '***Yes'*** and shut down the computer; then proceed to the next step.

Hardware Installation for Windows 98 Users

Plug-and-Play Devices

Personal488/PCI
Personal488/ATpnp
Personal488/Card

1. If you have not already done so, shutdown Windows 98 after the IEEE 488 software has been successfully installed.
2. Remove power from the PC.
3. Physically install your interface. As a quick reference,

   Personal488/PCI installs into a 32-bit PCI expansion slot,
   Personal488/ATpnp installs into a 16-bit ISA expansion slot, and
   Personal488/Card installs into a PC card slot.

4. Return power to the PC. Windows will detect your new hardware when the computer powers up.

   This completes the installation procedure.

"Non plug-and-play" Devices

Personal488/AT (AT488)
Personal488 (GP488B)
Personal488/MM

1. If you have not already done so, shutdown Windows 98 after the IEEE 488 software has been successfully installed.
2. Remove power from the PC.
3. Physically configure the device's jumpers and switches to match the resource settings Windows reported during the driver installation.

Non plug-and-play board users: physically configure your board's jumpers and switches to match the resource settings Windows reported. If these settings conflict with other hardware change the jumpers, switches, and Windows Resource settings to available resources.

Reference Note:
Refer to Chapter 4, *Hardware Configuration Reference* for further information concerning jumpers and switches.

4. Return power to the PC.

   This completes the installation procedure.

Notes

Windows Me Users

Software Installation

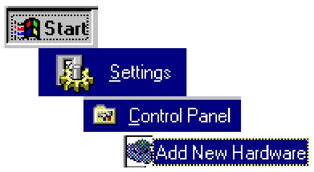
- For best results, install the interface after the software installation.
- If installing a second non plug-and-play interface, skip step 1.
- If installing a second plug-and-play interface, go to “Hardware Installation.”

Step 1

Insert the IEEE488 Software CD. The CD has an auto-run program that will automatically start the setup program when the CD is inserted into the CD ROM driver. If auto-run is disabled, use Explorer to launch the Setup.exe found in the root directory of the CD. Follow the screen prompts to install the software. Then if non plug-and-play hardware is being installed, proceed to step 2; otherwise proceed to *Hardware Installation for Windows Me Users* on page 3-19.

Step 2

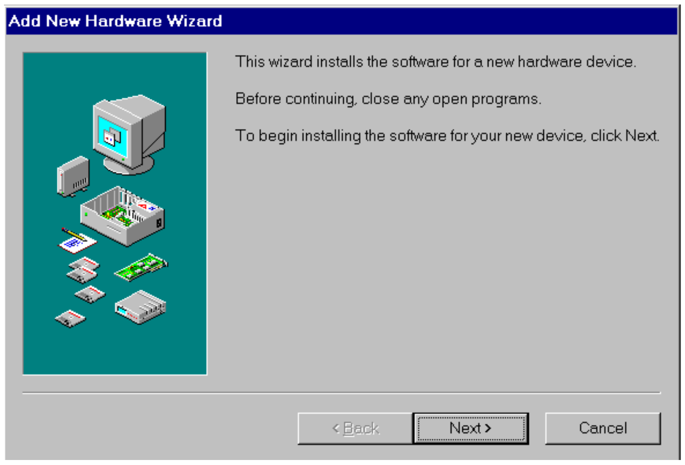
Use the “Add New Hardware” program found in the Control Panel to notify Windows Me that you are installing new hardware. Refer to the following steps that demonstrate the typical Windows panels encountered during the “Add New Hardware” program execution:



Start ⇒ Settings ⇒ Control Panel ⇒ Add New Hardware

Add New Hardware Procedure (*non plug-and-play users only*):

It is only necessary for users of “non plug-and-play” boards to follow the Add New Hardware Procedure. If your device is a “plug-and-play device,” skip this procedure.

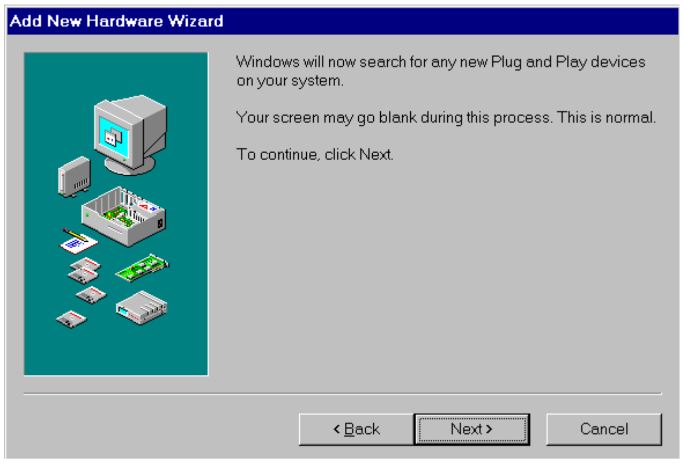


This wizard installs the software for a new hardware device.

Before continuing, close any open programs.

To begin installing the software for your new device, click Next.

1. The “Add New Hardware Wizard” displays an introductory message and prompts you to click ***Next***.

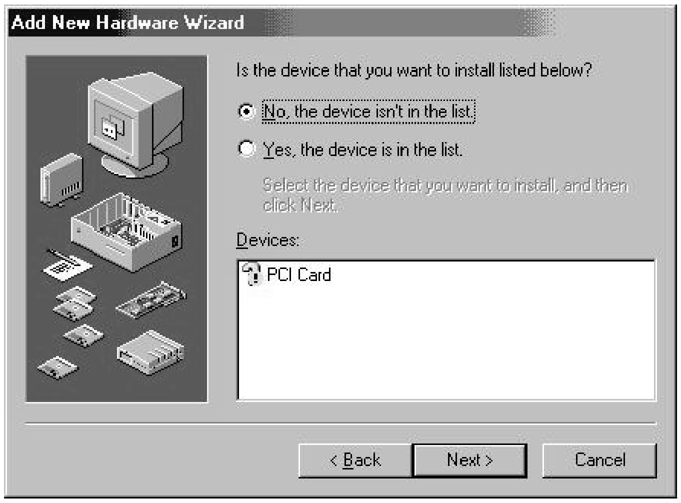


Windows will now search for any new Plug and Play devices on your system.

Your screen may go blank during this process. This is normal.

To continue, click Next.

2. Click ***Next***.



Is the device that you want to install listed below?

[✓ No, the device isn't in the list.]

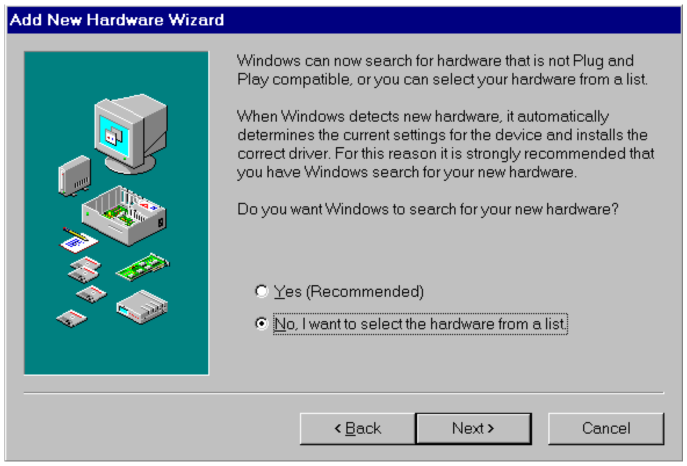
Yes, the device is in the list.

Select the device that you want to install, and then click Next.

Devices:

PCI Card

3. Select ‘***No, the device isn’t in the list***’ and click ***Next***.



Windows can now search for hardware that is not Plug and Play compatible, or you can select your hardware from a list.

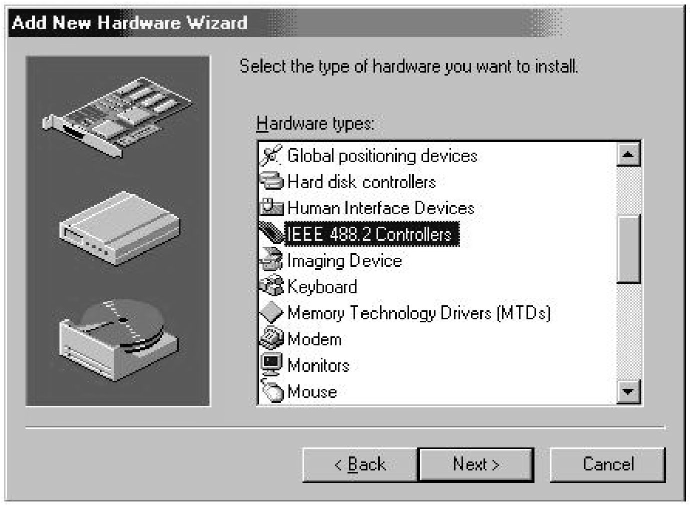
When Windows detects new hardware, it automatically determines the current settings for the device and installs the correct driver. For this reason it is strongly recommended that you have Windows search for your new hardware.

Do you want Windows to search for your new hardware?

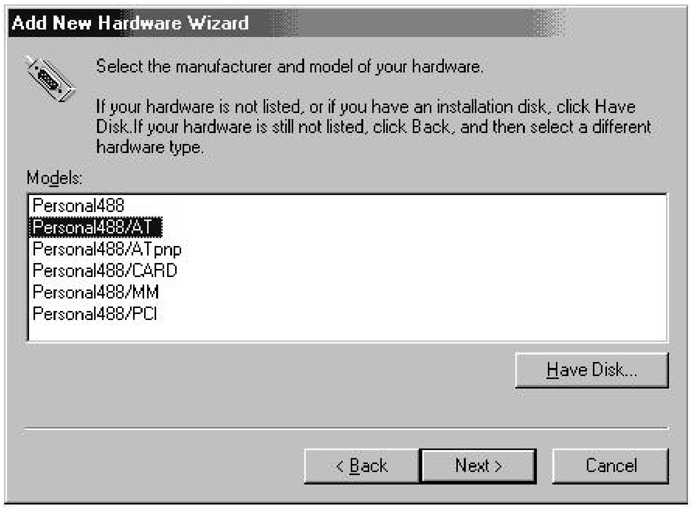
Yes (Recommended)

[✓ No, I want to select the hardware from a list.]

4. Select ‘***No, I want to select the hardware from a list’*** and click ***Next***.



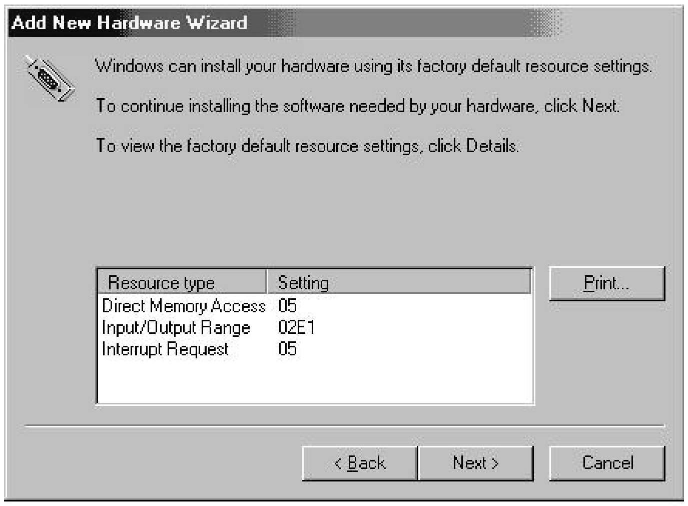
5. Select ‘***IEEE 488.2 Controllers’*** and click ***Next***.


Add New Hardware Wizard
Select the manufacturer and model of your hardware.
If your hardware is not listed, or if you have an installation disk, click Have Disk.If your hardware is still not listed, click Back, and then select a different hardware type.
Models:
Personal488
Personal488/AT
Personal488/ATpnp
Personal488/CARD
Personal488/MM
Personal488/PCI
Have Disk...
< Back
Next >
Cancel

6. Windows will now display a list of devices to install. **Select your specific Personal488 interface product.**

7. After making the selection, click ***Next***.

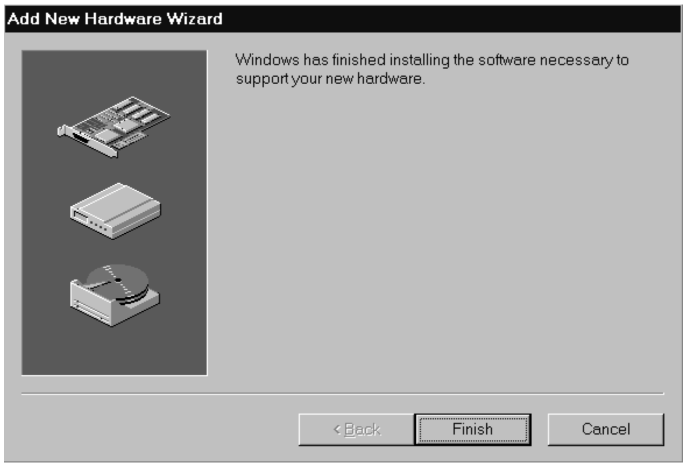
Windows will now display the default resource settings for your interface.


Add New Hardware Wizard
Windows can install your hardware using its factory default resource settings.
To continue installing the software needed by your hardware, click Next.
To view the factory default resource settings, click Details.
Print...
< Back
Next >
Cancel

Resource type	Setting
Direct Memory Access	05
Input/Output Range	02E1
Interrupt Request	05

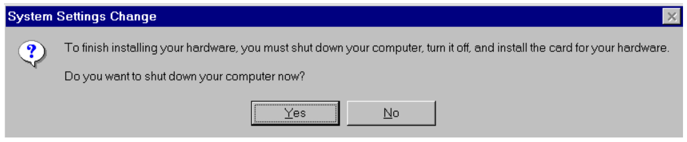
8. Make note of the displayed settings, as you must configure the jumpers and switch settings before installing an AT488 or GP488B.

Windows Me


Add New Hardware Wizard

Windows has finished installing the software necessary to support your new hardware.

9. Click ***Finish***.


System Settings Change

To finish installing your hardware, you must shut down your computer, turn it off, and install the card for your hardware.

Do you want to shut down your computer now?

10. Click ‘***Yes’*** and shut down the computer.

At this point, proceed to the next section, *Hardware Installation for Windows Me Users*.

Hardware Installation for Windows Me Users

Plug-and-Play Devices

Personal488/PCI
Personal488/ATpnp
Personal488/Card

1. If you have not already done so, shutdown Windows Me after the IEEE 488 software has been successfully installed.
2. Remove power from the PC.
3. Physically install your interface. As a quick reference,

   Personal488/PCI installs into a 32-bit PCI expansion slot,
   Personal488/ATpnp installs into a 16-bit ISA expansion slot, and
   Personal488/Card installs into a PC card slot.

4. Return power to the PC. After the computer powers up, Windows Me will detect your new hardware.

   This completes the installation procedure.

"Non plug-and-play" Devices

Personal488/AT (AT488)
Personal488 (GP488B)
Personal488/MM

1. If you have not already done so, shutdown Windows Me after the IEEE 488 software has been successfully installed.
2. Remove power from the PC.
3. Physically configure the device's jumpers and switches to match the resource settings Windows Me reported during the driver installation.

Non plug-and-play board users: physically configure your board's jumpers and switches to match the resource settings Windows reported. If these settings conflict with other hardware change the jumpers, switches, and Windows Resource settings to available resources.

Reference Note:
Refer to Chapter 4, *Hardware Configuration Reference* for further information concerning jumpers and switches.

4. Return power to the PC.

This completes the installation procedure.

Notes

Windows NT Users

Software Installation

Step 1

Insert the IEEE488 Software CD. The CD has an auto-run program that will automatically start the setup program when the CD is inserted into the CD ROM driver. If auto-run is disabled, use Explorer to launch the Setup.exe found in the root directory of the CD.

Step 2

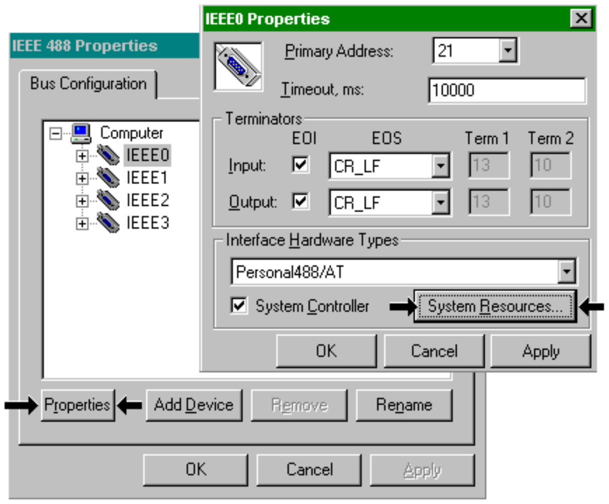
Follow the screen prompts to install the software, then proceed to hardware installation.

Hardware Installation

All Devices

Personal488/AT (AT488)
Personal488 (GP488Bplus)
Personal488/MM
Personal488/PCI

1. Access the Windows Control Panel and launch the *IEEE488* configuration program.
2. Highlight your interface (typically IEEE0) and select ***Properties*** as shown below:



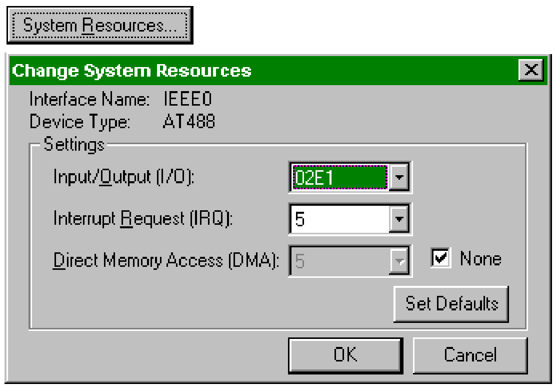
Primary Address: 21
Timeout, ms: 10000
Input: EOI ☑ EOS CR_LF Term 1 13 Term 2 10
Output: EOI ☑ EOS CR_LF Term 1 13 Term 2 10
Interface Hardware Types: Personal488/AT
☑ System Controller

Selecting "Properties" and "System Resources"

3. Click the ***Systems Resources*** button.

If Personal488/PCI is selected, then the System Resources button is not available. Windows NT is not a plug-and-play operating system, and thus changing resources on a Plug-and-Play interface is not possible.

Windows NT


System Resources...

Change System Resources

Interface Name: IEEE0

Device Type: AT488

Settings

Input/Output (I/O): 02E1

Interrupt Request (IRQ): 5

Direct Memory Access (DMA): 5 [✓ None]

Set Defaults

OK Cancel

4. Physically configure the device's jumpers and switches to match the resource settings Windows is reporting during the driver installation. If these settings conflict with other hardware, change the jumpers, switches, and Windows Resource settings to available resources.

Reference Note:
Refer to Chapter 4, *Hardware Configuration Reference*, for more information regarding jumpers and switches.

5. Shutdown Windows and remove power from your PC.
6. Insert the interface board, securing it appropriately.
7. Return power to the PC.

This completes the installation procedure.

Windows 2000 Users

Software Installation

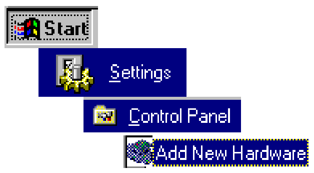
- For best results, install the interface after the software installation.
- If installing a second *non plug-and-play* interface, skip step 1.
- If installing a second *plug-and-play* interface, go to *Hardware Installation*, page 3-29.

Step 1

Insert the IEEE488 Software CD. The CD has an auto-run program that will automatically start the setup program when the CD is inserted into the CD ROM driver. If auto-run is disabled, use Explorer to launch the Setup.exe found in the root directory of the CD. Follow the screen prompts to install the software. Then, if non plug-and-play hardware is being installed, proceed to step 2; otherwise proceed to hardware installation on page 3-29.

Step 2

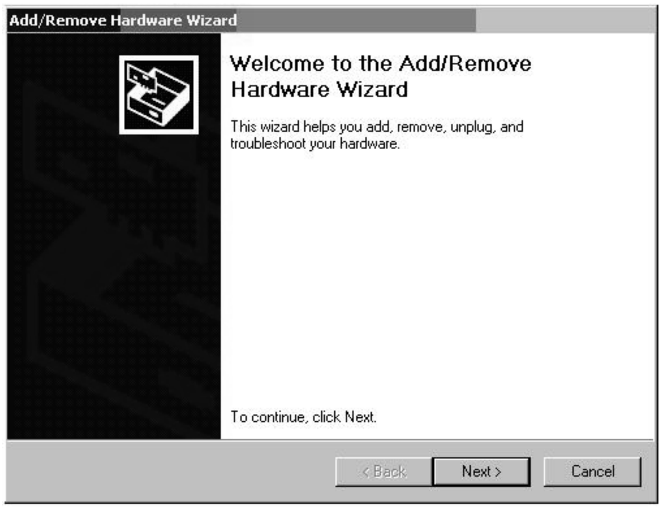
Use the "Add New Hardware" program found in the Control Panel to notify Windows 2000 that you are installing new hardware. Refer to the following steps that demonstrate the typical Windows panels encountered during the "Add New Hardware" program execution:



Start ⇒ Settings ⇒ Control Panel ⇒ Add New Hardware

Add New Hardware Procedure (*non plug-and-play users only*):

It is only necessary for users of "non plug-and-play" boards to follow the Add New Hardware Procedure. If your device is a "plug-and-play device," skip this procedure.



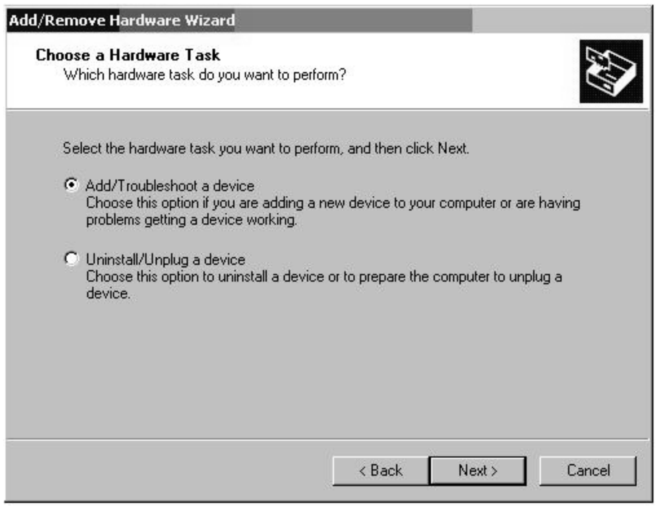
Welcome to the Add/Remove Hardware Wizard

This wizard helps you add, remove, unplug, and troubleshoot your hardware.

To continue, click Next.

1. After the Add/Remove Hardware Wizard appears, click ***Next***.

Windows 2000


Add/Remove Hardware Wizard

Choose a Hardware Task

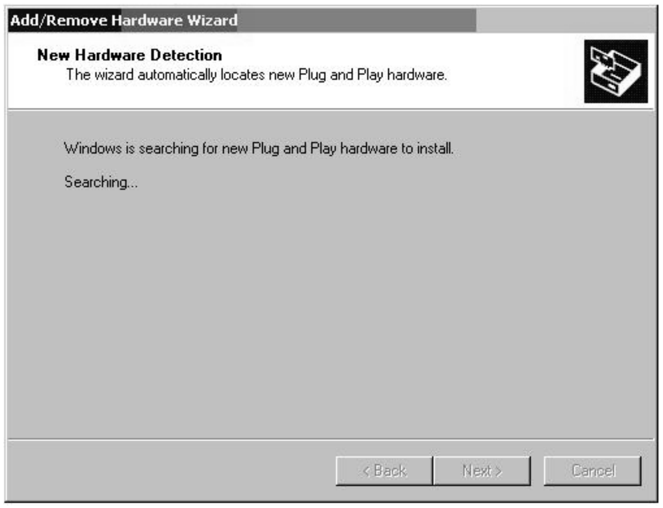
Which hardware task do you want to perform?

Select the hardware task you want to perform, and then click Next.

(●) Add/Troubleshoot a device
Choose this option if you are adding a new device to your computer or are having problems getting a device working.

( ) Uninstall/Unplug a device
Choose this option to uninstall a device or to prepare the computer to unplug a device.

2. Select **Add/Troubleshoot a device**.
3. Click ***Next***.


Add/Remove Hardware Wizard

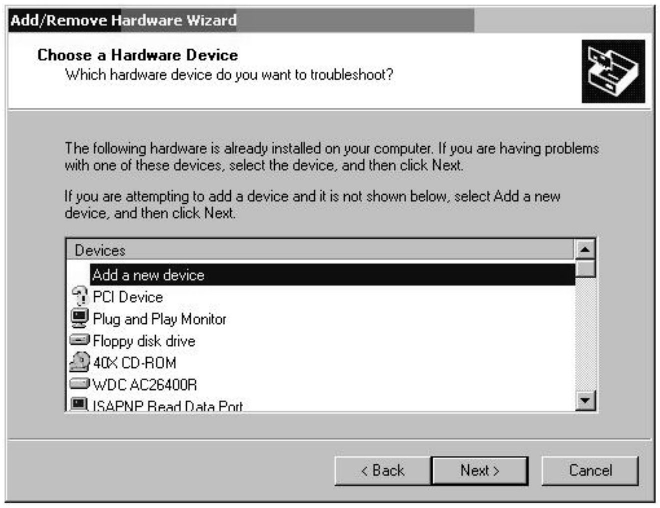
New Hardware Detection

The wizard automatically locates new Plug and Play hardware.

Windows is searching for new Plug and Play hardware to install.

Searching...

The *Add/Remove Hardware Wizard* searches for new plug-and-play hardware.


Add/Remove Hardware Wizard

Choose a Hardware Device

Which hardware device do you want to troubleshoot?

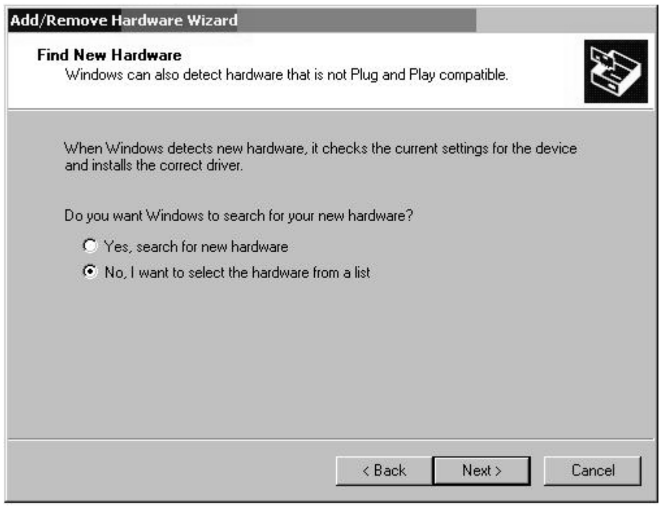
The following hardware is already installed on your computer. If you are having problems with one of these devices, select the device, and then click Next.

If you are attempting to add a device and it is not shown below, select Add a new device, and then click Next.

Devices
- Add a new device
- PCI Device
- Plug and Play Monitor
- Floppy disk drive
- 40X CD-ROM
- WDC AC26400R
- ISAPNP Read Data Port

After new hardware is located a screen, similar to that at the left, appears.

4. Select **Add a new device.**
5. Click ***Next***.


Add/Remove Hardware Wizard

Find New Hardware
Windows can also detect hardware that is not Plug and Play compatible.

When Windows detects new hardware, it checks the current settings for the device and installs the correct driver.

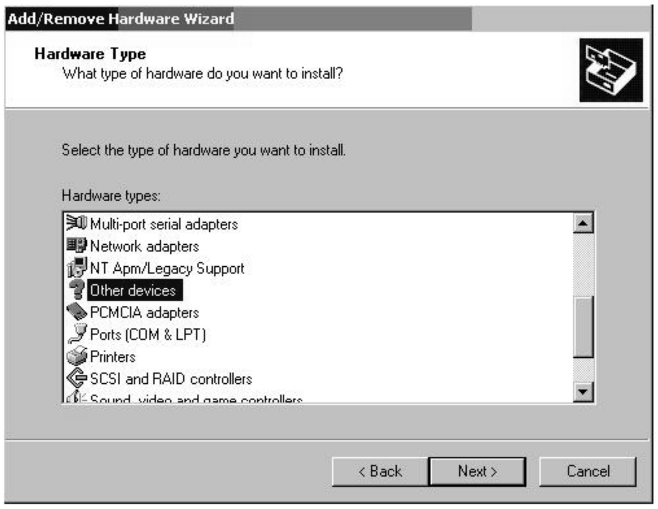
Do you want Windows to search for your new hardware?

( ) Yes, search for new hardware
[✓ No, I want to select the hardware from a list]

6. When prompted by the question, "Do you want Windows to search for your new hardware?"

   Select '***No, I want to select the hardware from a list***.'

7. Click ***Next***.


Add/Remove Hardware Wizard

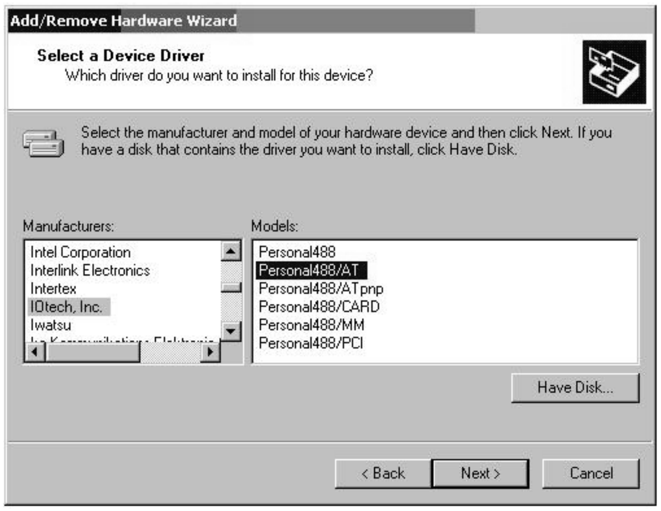
Hardware Type
What type of hardware do you want to install?

Select the type of hardware you want to install.

Hardware types:
Multi-port serial adapters
Network adapters
NT Apm/Legacy Support
[✓ Other devices]
PCMCIA adapters
Ports (COM & LPT)
Printers
SCSI and RAID controllers
Sound, video and game controllers

8. When asked what type of hardware you want to install, select **Other devices**.

9. Click ***Next***.


Add/Remove Hardware Wizard

Select a Device Driver
Which driver do you want to install for this device?

Select the manufacturer and model of your hardware device and then click Next. If you have a disk that contains the driver you want to install, click Have Disk.

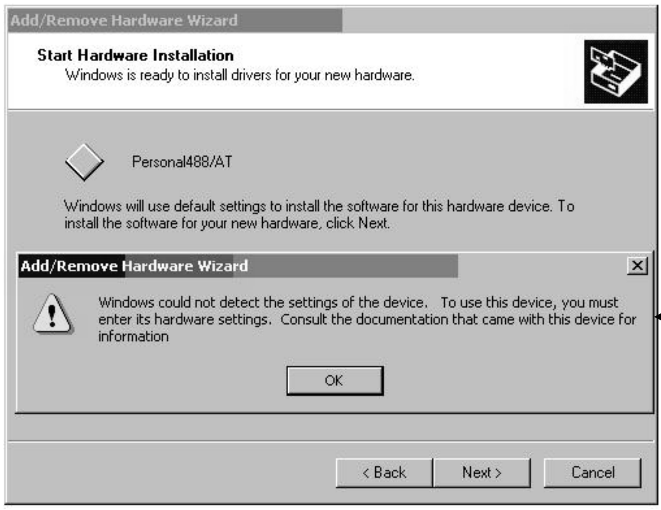
Manufacturers:
Intel Corporation
Interlink Electronics
Intertex
[✓ IOtech, Inc.]
Iwatsu

Models:
Personal488
[✓ Personal488/AT]
Personal488/ATpnp
Personal488/CARD
Personal488/MM
Personal488/PCI

Windows will display a list of manufacturers and device models.

11. Select **IOtech Inc**.
12. Select your specific Personal488 interface product.
13. Click ***Next***.

Windows 2000


Add/Remove Hardware Wizard

Start Hardware Installation

Windows is ready to install drivers for your new hardware.

Personal488/AT

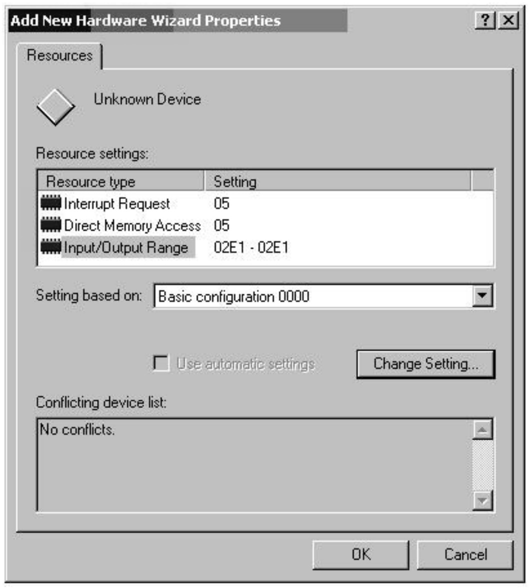
Windows will use default settings to install the software for this hardware device. To install the software for your new hardware, click Next.

Add/Remove Hardware Wizard

Windows could not detect the settings of the device. To use this device, you must enter its hardware settings. Consult the documentation that came with this device for information

14. Click ***Next***.
    Windows 2000 will install software for the device.

If Windows could not detect any hardware settings for the device, a message box informs you that hardware settings must be entered (see the following figure).


Add New Hardware Wizard Properties

Unknown Device

Resource settings:

Resource type	Setting
Interrupt Request	05
Direct Memory Access	05
Input/Output Range	02E1 - 02E1

Setting based on: Basic configuration 0000

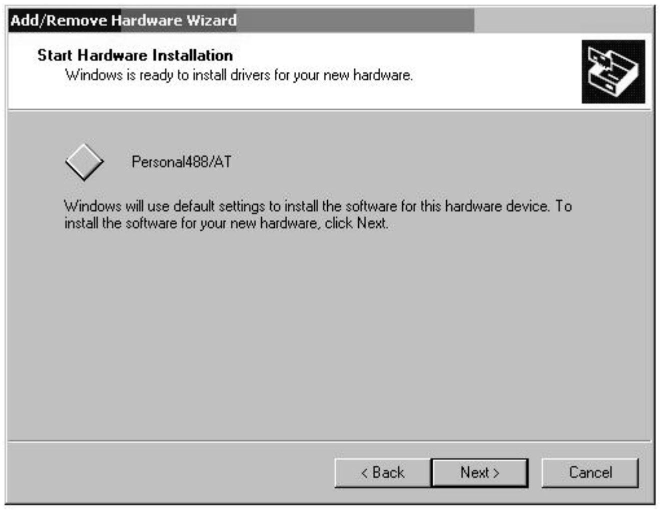
Use automatic settings

Conflicting device list:

No conflicts.

15. If Windows could not detect any hardware settings, as indicated by the message in step 14, enter the settings.

    Refer to *Hardware Configuration Reference* (chapter 4) for setting information.


Add/Remove Hardware Wizard

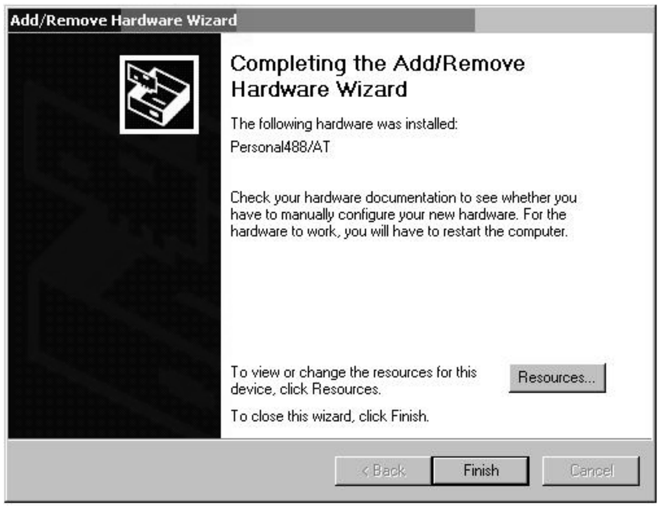
Start Hardware Installation

Windows is ready to install drivers for your new hardware.

Personal488/AT

Windows will use default settings to install the software for this hardware device. To install the software for your new hardware, click Next.

16. When prompted that "Windows is ready to install drivers for your new hardware," click ***Next***.



Completing the Add/Remove Hardware Wizard

The following hardware was installed:

Personal488/AT

Check your hardware documentation to see whether you have to manually configure your new hardware. For the hardware to work, you will have to restart the computer.

To view or change the resources for this device, click Resources.

To close this wizard, click Finish.

Windows will inform you that the hardware was installed.

17. Click **Finish**.

At this point proceed to the next section, *Hardware Installation for Windows 2000 Users.*

Notes

Hardware Installation for Windows 2000 Users

Plug-and-Play Devices

Personal488/PCI
Personal488/ATpnp
Personal488/Card

1. If you have not already done so, shutdown Windows 2000 after the IEEE 488 software has been successfully installed.
2. Remove power from the PC.
3. Physically install your interface. As a quick reference,

   Personal488/PCI installs into a 32-bit PCI expansion slot,
   Personal488/ATpnp installs into a 16-bit ISA expansion slot, and
   Personal488/Card installs into a PC card slot.

4. Return power to the PC. After the computer powers up, Windows 2000 will detect your new hardware.

   This completes the installation procedure.

"Non plug-and-play" Devices

Personal488/AT (AT488)
Personal488 (GP488B)
Personal488/MM

1. If you have not already done so, shutdown Windows 2000 after the IEEE 488 software has been successfully installed.
2. Remove power from the PC.
3. Physically configure the device's jumpers and switches to match the resource settings Windows 2000 reported during the driver installation.

Non plug-and-play board users: physically configure your board's jumpers and switches to match the resource settings Windows reported. If these settings conflict with other hardware change the jumpers, switches, and Windows Resource settings to available resources.

Reference Note:
Refer to Chapter 4, *Hardware Configuration Reference* for further information concerning jumpers and switches.

4. Return power to the PC.

This completes the installation procedure.

Windows 2000

Notes

Hardware Configuration Reference 4

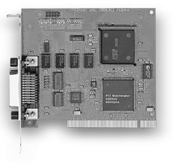
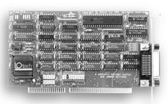
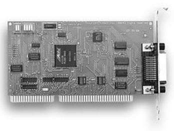
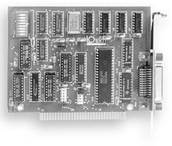
Hardware Configuration

Plug-and-Play Devices

Plug-and-play devices require no physical configuration of hardware. After installing your software and hardware [as described in Chapters 2 and 3] the configuration is performed automatically. Note that the *plug-and-play* devices are listed in the following table as a product reference only. This chapter contains no useful information concerning *plug-and-play* devices.

Non Plug-and-Play Devices

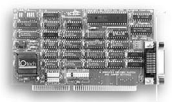
The I/O base address, IRQ, and DMA settings of ***non plug-and-play*** devices are determined by the physical settings of jumpers and DIP switches. This chapter provides the information necessary to configure these devices.

Plug-and-Play Devices		Non Plug-and-Play Devices	
PCI488 	Automatic Configuration.	**AT488** 	See page 4-3.
AT488pnp 	Automatic Configuration.	**GP488B** 	See page 4-7.
CARD488	Automatic Configuration.	**GP488B/MM**	See page 4-11.

Note: The device images are not shown to the same scale.

Notes

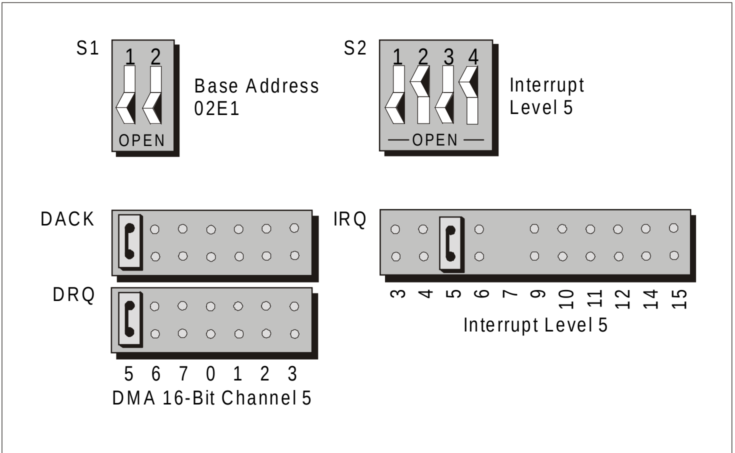
AT488 Configurations



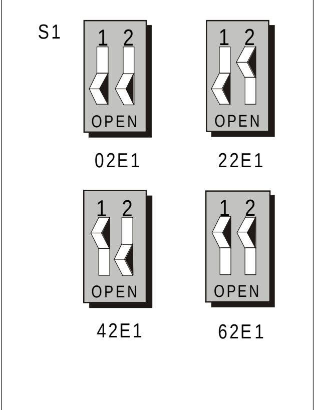
The I/O base address, IRQ, and DMA settings are switch/jumper selectable via the following locations on the AT488 interface board: One 2-microswitch DIP switch labelled S1, one 4-microswitch DIP switch labelled S2, two 14-pin headers labelled DACK and DRQ, and one 22-pin header labelled IRQ. The DIP switch settings, and the arrangement of the jumpers on the headers set the hardware configuration.

For the next steps, make sure that the I/O address, IRQ, and DMA set on the interface board are different from any existing ports in your system. A conflict results when two I/O addresses, IRQs, or DMAs are the same. (As the exception, additional AT488 interfaces may share the same IRQ and DMA values.) If there is a conflict, reconfigure the switch/jumper settings. Refer to the following figures as needed.

Configuring the AT488 Interface I/O Base Address



AT488 Default Settings



AT488 I/O Base Address Selections

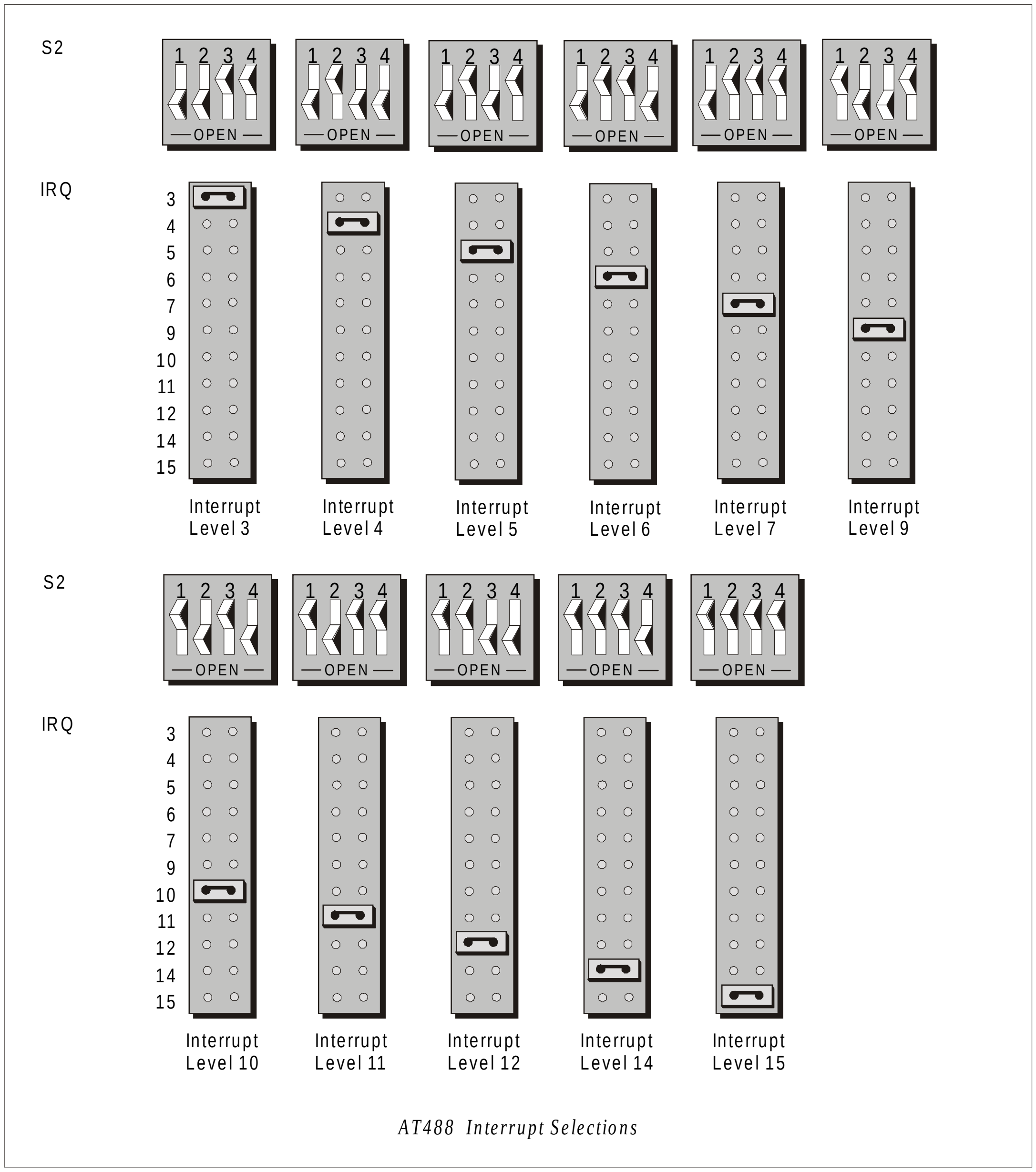
The factory default I/O base address is **`02E1`**. If this creates a conflict, reset switch S1 according to the figure and following table. The register addresses will be automatically relocated at fixed offsets from the base address. If reset, record the new Input/Output (I/O) address being used.

Selected I/O Base Address				Register	
02E1	**22E1**	**42E1**	**62E1**		
Automatic Offset Addresses				**Read Register**	**Write Register**
02E1	**22E1**	**42E1**	**62E1**	Data In	Data Out
06E1	**26E1**	**46E1**	**66E1**	Interrupt Status 1	Interrupt Mask 1
0AE1	**2AE1**	**4AE1**	**6AE1**	Interrupt Status 2	Interrupt Mask 2
0EE1	**2EE1**	**4EE1**	**6EE1**	Serial Poll Status	Serial Poll Mode
12E1	**32E1**	**52E1**	**72E1**	Address Status	Address Mode
16E1	**36E1**	**56E1**	**76E1**	CMD Pass Through	Auxiliary Mode
1AE1	**3AE1**	**5AE1**	**7AE1**	Address 0	Address 0/1
1EE1	**3EE1**	**5EE1**	**7EE1**	Address 1	End of String

The I/O base address sets the addresses used by the computer to communicate with the IEEE 488 interface hardware on the board. The address is normally specified in hexadecimal and can be **`02E1`**, **`22E1`**, **`42E1`**, or **`62E1`**. The registers of the IOT7210 IEEE 488 controller chip and other auxiliary registers are then located at fixed offsets from the base address.

Most versions of Driver488 are capable of managing as many as four IEEE 488 interfaces. To do so, the interface configurations must be arranged to avoid conflict amongst themselves. No two boards may have the same I/O address; but they may, and usually should, have the same DMA channel and interrupt level.

Configuring the AT488 Interface Interrupt (IRQ)



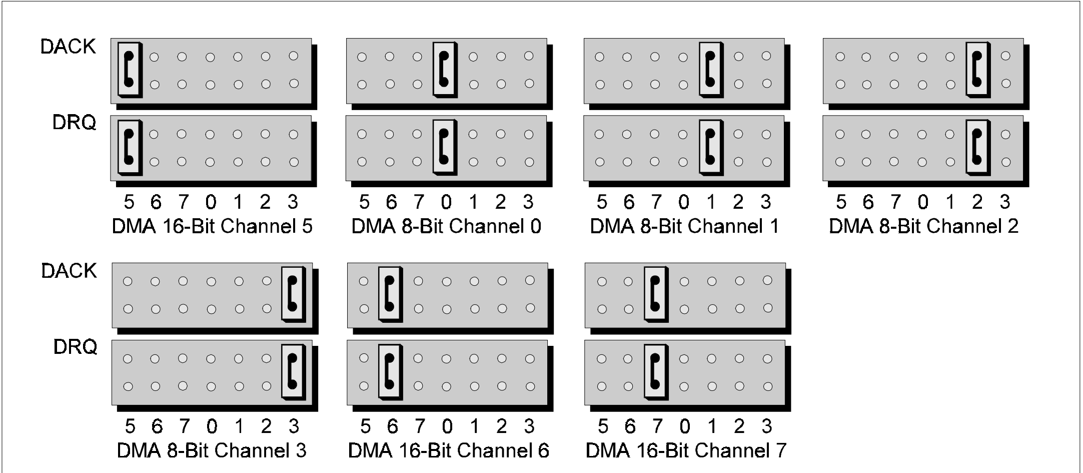
AT488 Interrupt Selections

The factory default Interrupt (IRQ) is 5. If this creates a conflict, reset switch S2 and jumper IRQ according to the figure. The switch and jumper settings must both indicate the same interrupt level for correct operation with interrupts. If reset, record the new Interrupt (IRQ) being used.

The AT488 interface board may be set to interrupt the PC on the occurrence of certain hardware conditions. The main board interrupt may be set to IRQ level 3 through 7, 9 through 12, 14, or 15. Interrupts 10 through 15 are only available in a 16-bit slot on an AT-class machine. Interrupt 9 becomes synonymous with Interrupt 2 when used in a PC/XT bus.

The selected interrupt may be shared among several AT488s in the same PC/AT chassis. The AT488 adheres to the "AT-style" interrupt sharing conventions. When the AT488 requires service, the IRQ jumper determines which PC interrupt level is triggered. When an interrupt occurs, the interrupting device must be reset by writing to an I/O address which is different for each interrupt level. The switch settings determine the I/O address to which the board's interrupt rearm circuitry responds.

Configuring the AT488 Interface DMA Channel



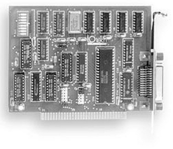
AT488 DMA Channel Selections

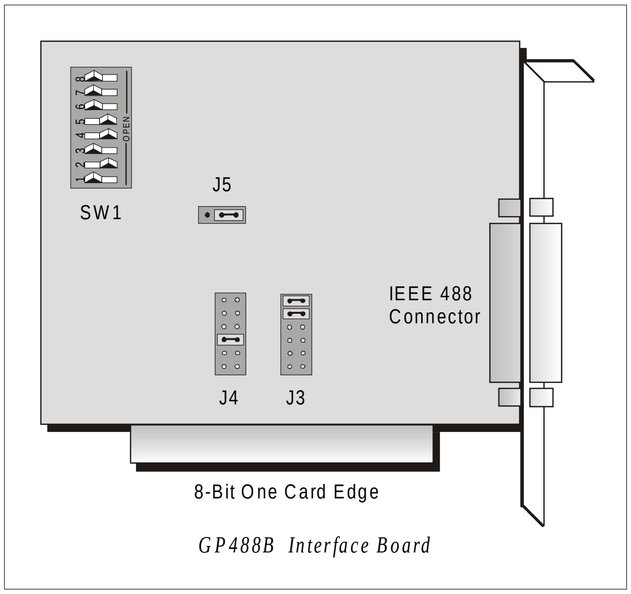
The factory default DMA channel is 5. If this creates a conflict, reset jumpers DACK and DRQ according to the figure. Both the DRQ and DACK jumpers must be set to the desired DMA channel for proper operation. If reset, record the new DMA channel being used.

Direct Memory Access (DMA) is a high-speed method of transferring data from or to a peripheral, such as a digitizing oscilloscope, to or from the PC's memory. The AT class machine has seven DMA channels. Channels 0 to 3 (8-bit), 5, 6, and 7 (16-bit) are available only in a 16-bit slot on a PC/AT-class machine. Channel 2 is usually used by the floppy disk controller, and is unavailable. Channel 3 is often used by the hard disk controller in PCs, XTs, and the PS/2 with the ISA bus, but is usually not used in ATs. Channels 5 to 7 are 16-bit DMA channels and offer the highest throughput (up to 1 Megabyte per second). Channels 0 to 3 are 8-bit DMA channels and although slower, they offer compatibility with existing GP488B and GP488B/MM applications that only made use of 8-bit DMA channels. Under some rare conditions, it is possible for high-speed transfers on DMA Channel 1 to demand so much of the available bus bandwidth that simultaneous access of a floppy controller will be starved for data due to the relative priorities of the two channels.

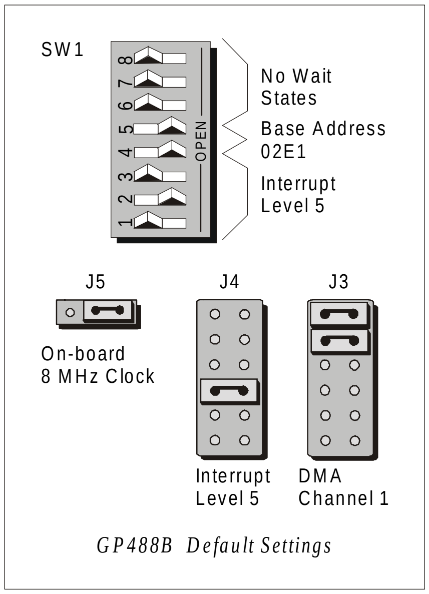
Notes

GP488B Configurations





GP488B Interface Board



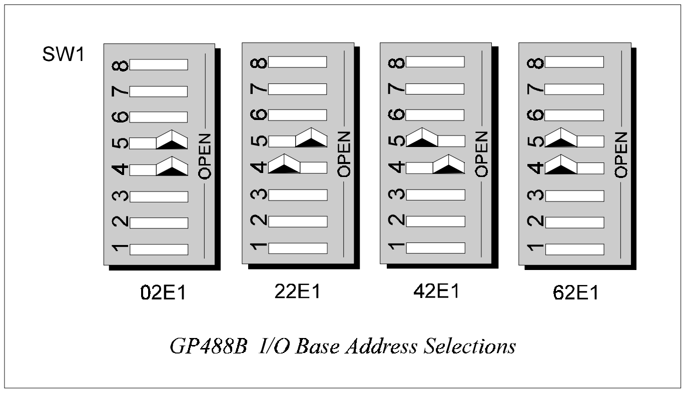
GP488B Default Settings

The I/O base address, IRQ, and DMA settings are switch/jumper selectable via the following locations on the GP488B interface board: One 8-microswitch DIP switch labelled SW1, two 12-pin headers labelled J3 and J4, and one 3-pin header labelled J5. The DIP switch settings, and the arrangement of the jumpers on the headers set the hardware configuration.

For the next steps, make sure that the I/O address, IRQ, and DMA, set on the interface board are different from any existing ports in your system. A conflict results when two I/O addresses, IRQs, or DMAs are the same. (As the exception, additional GP488B interfaces may share the same IRQ and DMA values). If there is a conflict, reconfigure the switch/jumper settings. Refer to the following figures as needed.

Selected I/O Base Address				Register	
02E1	22E1	42E1	62E1		
Automatic Offset Addresses				**Read Register**	**Write Register**
02E1	22E1	42E1	62E1	Data In	Data Out
06E1	26E1	46E1	66E1	Interrupt Status 1	Interrupt Mask 1
0AE1	2AE1	4AE1	6AE1	Interrupt Status 2	Interrupt Mask 2
0EE1	2EE1	4EE1	6EE1	Serial Poll Status	Serial Poll Mode
12E1	32E1	52E1	72E1	Address Status	Address Mode
16E1	36E1	56E1	76E1	CMD Pass Through	Auxiliary Mode
1AE1	3AE1	5AE1	7AE1	Address 0	Address 0/1
1EE1	3EE1	5EE1	7EE1	Address 1	End of String

Configuring the GP488B Interface I/O Base Address



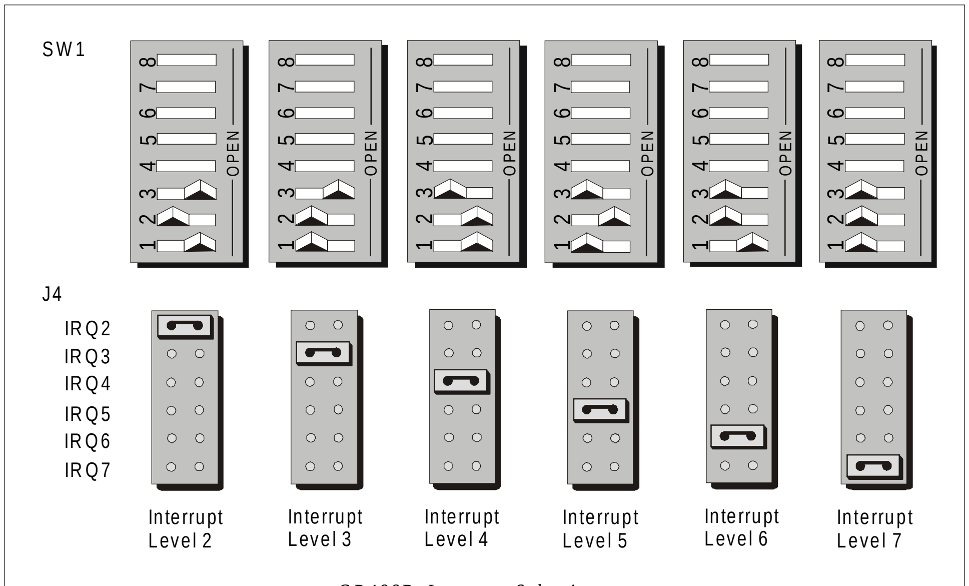
GP488B I/O Base Address Selections

he factory default I/O base address is **`02E1`**. If this creates a conflict, reset SW1 microswitches 4 and 5 ccording to the figure and following table. The register addresses will be automatically relocated at fixed ffsets from the base address. If reset, record the new Input/Output (I/O) address being used.

The I/O base address sets the addresses used by the computer to communicate with the IEEE 488 interface hardware on the board. The address is normally specified in hexadecimal and can be **`02E1`**, **`22E1`**, **`42E1`**, or **`62E1`**. The registers of the IOT7210 IEEE 488 controller chip and other auxiliary registers are then located at fixed offsets from the base address.

Most versions of Driver488 are capable of managing as many as four IEEE 488 interfaces. To do so, the interface configurations must be arranged to avoid conflict amongst themselves. No two boards may have the same I/O address; but they may, and usually should, have the same DMA channel and interrupt level.

Configuring the GP488B Interface Interrupt (IRQ)

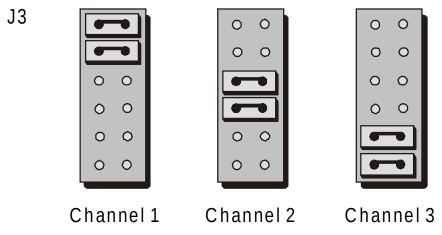


GP488B Interrupt Selections

The factory default Interrupt (IRQ) is 5. If this creates a conflict, reset SW1 microswitches 1, 2, and 3, and jumper J4 according to the figure. The switch and jumper settings must both indicate the same interrupt level for correct operation with interrupts. If reset, record the new Interrupt (IRQ) being used.

The GP488B interface board may be set to interrupt the PC on the occurrence of certain hardware conditions. The level of the interrupt generated is set by J4. The GP488B adheres to the “AT-style” interrupt sharing conventions. When an interrupt occurs, the interrupting device must be reset by writing to I/O address **`02FX`**, where X is the interrupt level (from 0 to 7). This interrupt response level is set by switches 1, 2, and 3 of SW1 which must be set to correspond to the J4 interrupt level setting.

Configuring the GP488B Interface DMA Channel



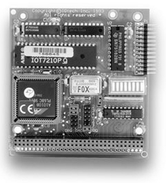
GP488B DMA Channel Selections

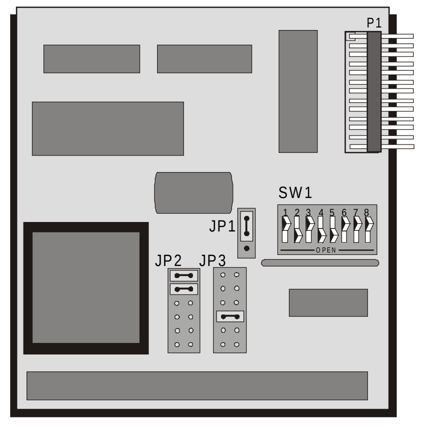
The factory default DMA channel is 1. If this creates a conflict, reset jumper J3 according to the figure. If reset, record the new DMA channel being used.

Direct Memory Access (DMA) is a high-speed method of transferring data from or to a peripheral, such as a digitizing oscilloscope, to or from the PC’s memory. The PC has four DMA channels, but Channel 0 (Disabled) is used for memory refresh and is not available for peripheral data transfer. Channel 2 is usually used by the floppy disk controller, and is also unavailable. Channel 3 is often used by the hard disk controller in PCs, XTs, and the PS/2 with the ISA bus, but is usually not used in ATs. So, depending on your hardware, DMA Channels 1 and possibly Channel 3 are available. Under some rare conditions, it is possible for high-speed transfers on DMA Channel 1 to demand so much of the available bus bandwidth that simultaneous access of a floppy controller will be starved for data due to the relative priorities of the two channels.

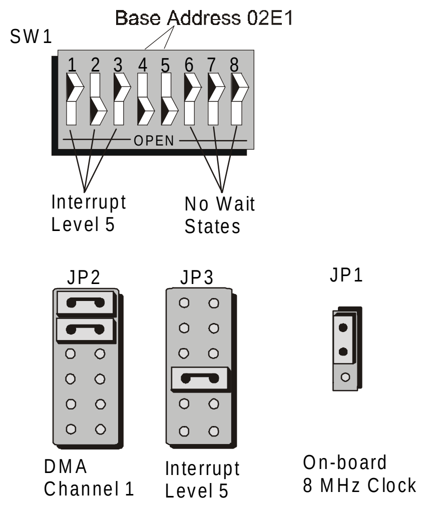
Notes

GP488B/MM PC104 Configurations





GP488B/MM PC104 Interface Board



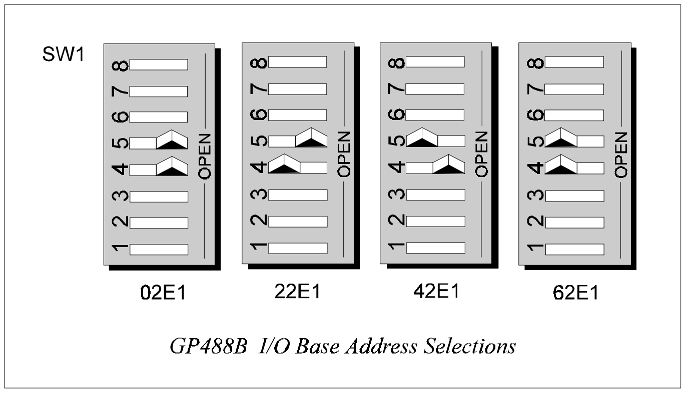
GP488B/MM PC104 Default Settings

The I/O base address, IRQ, and DMA settings are switch/jumper selectable via the following locations on the GP488B/MM interface board: One 8-microswitch DIP switch labelled SW1, two 12-pin headers labeled JP2 and JP3, and one 3-pin header labeled JP1. The DIP switch settings, and the arrangement of the jumpers on the headers set the hardware configuration.

For the next steps, make sure that the I/O address, IRQ, and DMA, set on the interface board are different from any existing ports in your system. A conflict results when two I/O addresses, IRQs, or DMAs are the same. (As the exception, additional GP488B/MM interfaces may share the same IRQ and DMA values). If there is a conflict, reconfigure the switch/jumper settings. Refer to the following figures as needed.

Selected I/O Base Address				Register	
02E1	22E1	42E1	62E1		
Automatic Offset Addresses				**Read Register**	**Write Register**
02E1	22E1	42E1	62E1	Data In	Data Out
06E1	26E1	46E1	66E1	Interrupt Status 1	Interrupt Mask 1
0AE1	2AE1	4AE1	6AE1	Interrupt Status 2	Interrupt Mask 2
0EE1	2EE1	4EE1	6EE1	Serial Poll Status	Serial Poll Mode
12E1	32E1	52E1	72E1	Address Status	Address Mode
16E1	36E1	56E1	76E1	CMD Pass Through	Auxiliary Mode
1AE1	3AE1	5AE1	7AE1	Address 0	Address 0/1
1EE1	3EE1	5EE1	7EE1	Address 1	End of String

Configuring the GP488B/MM PC104 Interface I/O Base Address



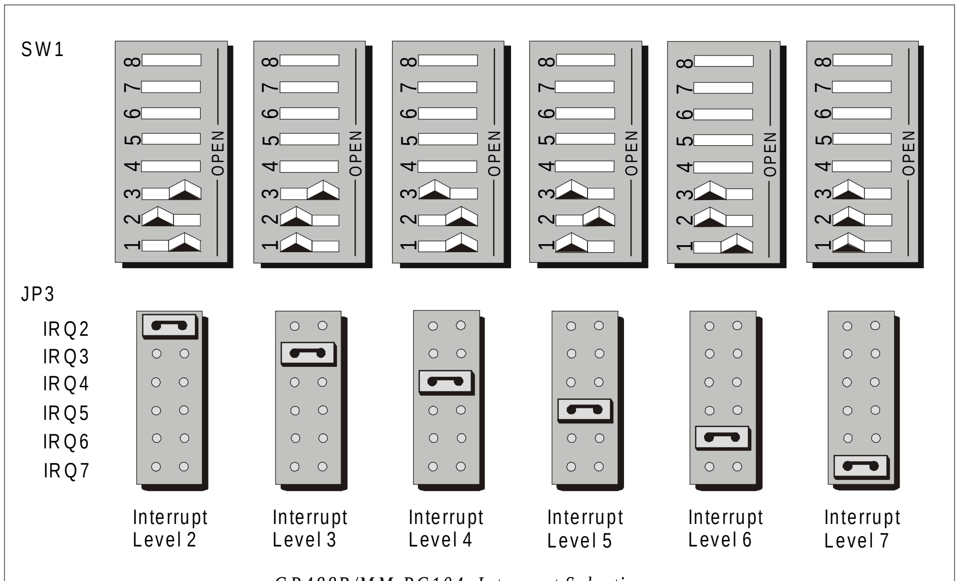
GP488B I/O Base Address Selections

The factory default I/O base address is **`02E1`**. If this creates a conflict, reset SW1 microswitches 4 and 5 according to the figure and following table. The register addresses will be automatically relocated at fixed offsets from the base address. If reset, record the new Input/Output (I/O) address being used.

The I/O base address sets the addresses used by the computer to communicate with the IEEE 488 interface hardware on the board. The address is normally specified in hexadecimal and can be **`02E1`**, **`22E1`**, **`42E1`**, or **`62E1`**. The registers of the IOT7210 IEEE 488 controller chip and other auxiliary registers are then located at fixed offsets from the base address.

Most versions of Driver488 are capable of managing as many as four IEEE 488 interfaces. To do so, the interface configurations must be arranged to avoid conflict. No two boards may have the same I/O address; but they may, and usually should, have the same DMA channel and interrupt level.

Configuring the GP488B/MM PC104 Interface Interrupt (IRQ)

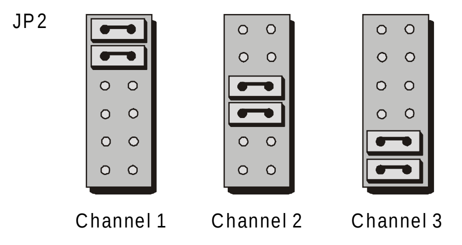


GP488B/MM PC104 Interrupt Selections

The factory default Interrupt (IRQ) is 5. If this creates a conflict, reset SW1 microswitches 1, 2, and 3, and jumper JP3 according to the figure. The switch and jumper settings must both indicate the same interrupt level for correct operation with interrupts. If reset, record the new Interrupt (IRQ) being used.

The GP488B/MM interface board may be set to interrupt the PC on the occurrence of certain hardware conditions. The level of the interrupt generated is set by JP3. The GP488B/MM adheres to the “AT-style” interrupt sharing conventions. When an interrupt occurs, the interrupting device must be reset by writing to I/O address **`02FX`**, where X is the interrupt level (from 0 to 7). This interrupt response level is set by switches 1, 2, and 3 of SW1 which must be set to correspond to the JP3 interrupt level setting.

Configuring the GP488B/MM PC104 Interface DMA Channel



GP488B/MM PC104 DMA Channel Selections

The factory default DMA channel is 1. If this creates a conflict, reset jumper JP2 according to the figure. If reset, record the new DMA channel being used.

Direct Memory Access (DMA) is a high-speed method of transferring data from or to a peripheral, such as a digitizing oscilloscope, to or from the PC’s memory. The PC has four DMA channels, but Channel 0 (Disabled) is used for memory refresh and is not available for peripheral data transfer. Channel 2 is usually used by the floppy disk controller, and is also unavailable. Channel 3 is often used by the hard disk controller in PCs, XTs, and the PS/2 with the ISA bus, but is usually not used in ATs. So, depending on your hardware, DMA Channels 1 and possibly Channel 3 are available. Under some rare conditions, it is possible for high-speed transfers on DMA Channel 1 to demand so much of the available bus bandwidth that simultaneous access of a floppy controller will be starved for data due to the relative priorities of the two channels.

Notes

Using IEEE 488 5

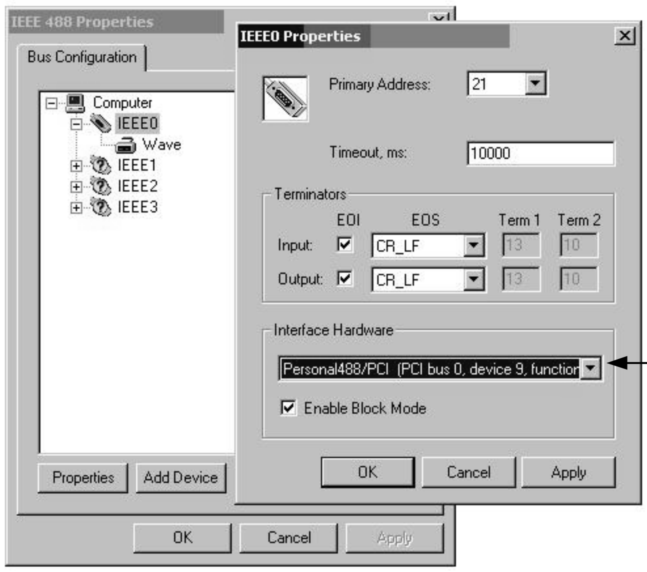
IEEE 488 Configuration Utility

The IEEE 488 configuration utility is located in Windows Control Panel. The configuration utility is primarily used to setup the Interfaces and External Devices. As seen in the following figure, it is similar to the Device Manager.

Note: By default, the configuration utility always shows four interfaces (IEEE0, IEEE 1, IEEE 2, & IEEE 3), even when only one interface is actually installed.

Interfaces

The minimum requirement for configuring your system is to make certain the IEEE 488.2 interface is properly selected and configured.



Primary Address: 21
Timeout, ms: 10000

	EOI	EOS	Term 1	Term 2
Input:	☑	CR_LF	13	10
Output:	☑	CR_LF	13	10

Personal488/PCI (PCI bus 0, device 9, function
☑ Enable Block Mode

The Interface Hardware box lists all the available interfaces. In the figure, IEEE0 is assigned to a Personal488/PCI interface.

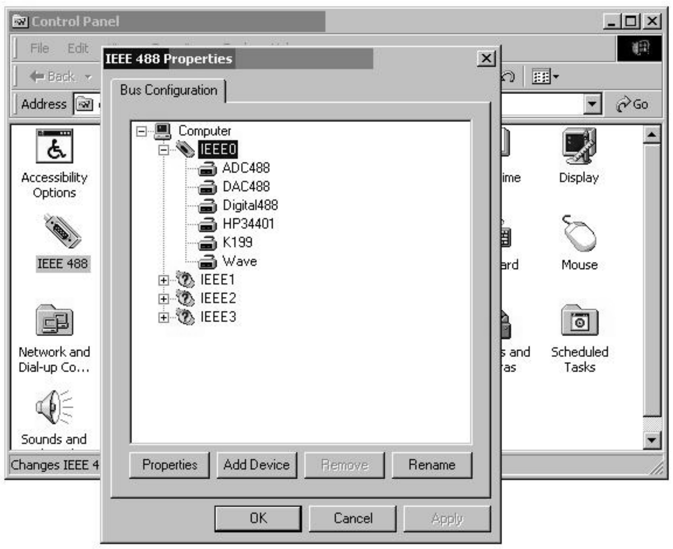
External Devices

Within your IEEE 488.2 application program, a handle accesses devices on the bus. Accessing *named devices* creates these handles. For example, the following function call:
handle = OpenName("Wave")
returns a handle for a device named Wave.

Each device handle is a means of maintaining a record of the following three configurable parameters:

- IEEE 488 address
- IEEE 488 bus terminators (EOS characters)
- associated time-out period

To easily create named devices, you can use the IEEE 488 configuration utility. An alternative is to use API function calls to configure devices within an application itself. Using the API function calls to make and configure external devices eliminates the need to run the IEEE 488 configuration utility every time the application program is installed onto a different PC.



The IEEE 488 program is located in the Windows Control Panel. IEEE 488 control buttons consist of the following:

Properties: launches the properties panel for the selected device.

Add Device: Adds a new generic device.

Remove: Removes any external devices. IEEE interfaces cannot be removed.

Rename: Allows renaming of external devices. Typically, external devices are named after their manufacture, model, or function.

Configuration Parameters

Name: External device names are used to convey the configuration information about each device. The name is used to obtain a handle to that device which will be used by all the API function calls. External device names can consist of 1 to 32 characters, and the first character must be a letter. The remaining characters may be letters, numbers, or underscores (_). External device names are **NOT** case sensitive.
IEEE Bus Address: This is the setting for the IEEE 488 bus address of the board. The IEEE 488 address consists of a primary address from 00 to 30, and an optional secondary address from 00 to 31.

When a device has multiple secondary addresses, it may be useful to have several different external device names defined for such a device. In this case, an array of handles could be maintained for easy access to different IEEE 488 addresses throughout the application.

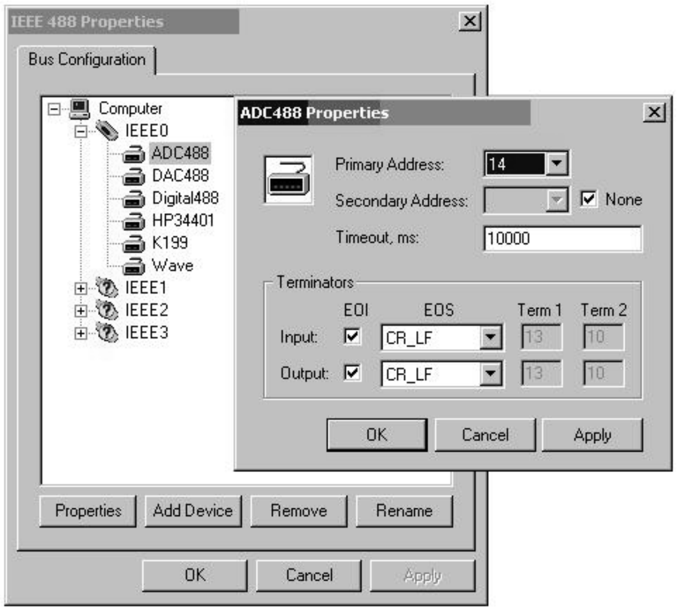
Timeout (ms): The time out period is the amount of time data transfers wait before reporting a time out error. If the time out period elapses while waiting for data to be transferred or while transferring data, an error is generated. The default value for the time out period is 10 seconds (10000 milliseconds).

Bus Terminators: The IEEE 488 bus terminators specify the characters and/or End-or-Identify (EOI) signal that is to be appended to data that is sent to the external device, or mark the end of data that is received from the external device.

Incorrectly configured bus terminators often cause many subtle problems. Many newer IEEE 488.2 instruments have standardized on the Line Feed <LF> character, however older instruments were free to use any character or character combination. When in doubt, refer to the instrument's programming manual or contact the manufacturer.

Each device name is a means of maintaining a record of three configurable parameters:

- IEEE 488 address
- time out period
- EOS terminators



Primary Address: 14
Secondary Address: None ✓
Timeout, ms: 10000

Terminators

	EOI	EOS	Term 1	Term 2
Input:	✓	CR_LF	13	10
Output:	✓	CR_LF	13	10

WinTest – Driver488 Workshop

This section pertains to the Wintest utility program. After reviewing the material you should be able to use Wintest to verify your Driver488 installation and to perform simple communication with your IEEE 488 instruments.

What is Wintest?

Wintest is a utility program included with the Personal488 packages. Its primary application is to exercise the driver by communicating with instruments on the IEEE 488 bus. Communication is accomplished using API function calls, which are accessible via the menu bar. Every API function can be exercised using Wintest. Under the menu items of the Wintest application are all of the API functions calls. Each menu item represents a category, which are further described below:

Menu Item	Group Description
Device	Commands dealing with accessing and configuring instruments
Enter	Used to read data from an instrument.
Output	Used to write data or device-dependent setup commands to an instrument.
Send	Low level IEEE 488 commands (LAG, TAG, UNL UNT, etc)
Query	Status polling commands
Error	Driver query and error handling commands
Events	Commands dealing with enabling and disabling bus events
Bus	Bus and instrument management commands
Config	Driver configuration commands

Why use Wintest?

Wintest can be used to exercise and test every available API command. In addition, since Wintest displays the command's function call, it is useful for example purposes.

Who uses Wintest?

The Wintest is an interactive utility intended for individuals:

- New to IOtech's Personal488 IEEE 488 controllers
- New to IEEE 488 control
- Getting familiar with IEEE 488 instruments
- Performing quick communication tests on instruments
- Performing installation verification

Where is Wintest located?

Wintest is located in the '**Applications**' folder found under the installation directory. If you choose the default directory Wintest will be located in

C:\Program Files\IEEE488\Applications\

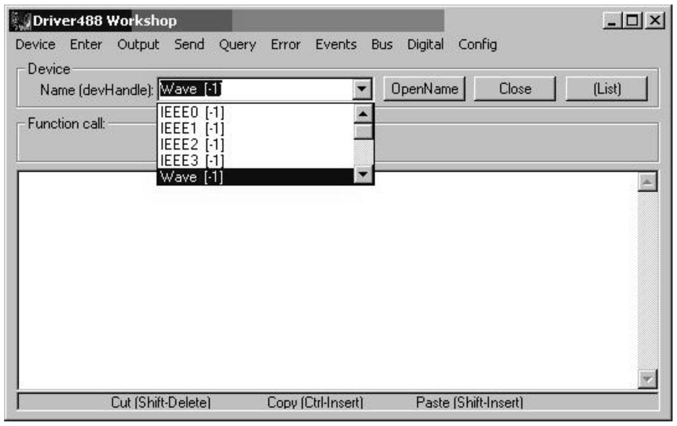
The installation process sets up a shortcut to Wintest. This shortcut is found under

Start=>Programs =>IOtech Driver488=>Wintest

The shortcut can be used to easily start Wintest.

How can I verify that my Personal488 interface is installed and working?

This section consists of a 6-step walk-through in which Wintest is used to verify instrument communication.

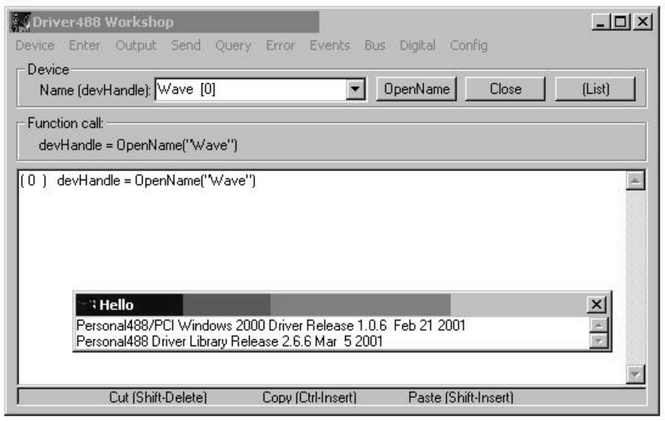

Driver488 Workshop
Device Enter Output Send Query Error Events Bus Digital Config
Device
Name (devHandle): Wave [-1]
OpenName
Close
(List)
IEEE0 [-1]
IEEE1 [-1]
IEEE2 [-1]
IEEE3 [-1]
Wave [-1]
Function call:
Cut (Shift-Delete) Copy (Ctrl-Insert) Paste (Shift-Insert)

Step 1

Launch ***Wintest*** from the ***Programs*** menu.

Step 2

Select 'Wave[-1]' from the ***devHandle*** drop down box.


Driver488 Workshop
Device Enter Output Send Query Error Events Bus Digital Config
Device
Name (devHandle): Wave [0]
OpenName
Close
(List)
Function call:
Hello
Cut (Shift-Delete) Copy (Ctrl-Insert) Paste (Shift-Insert)

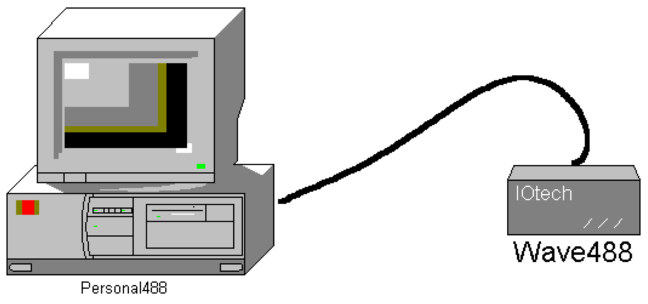
```
devHandle = OpenName("Wave")
( 0 )  devHandle = OpenName("Wave")
Personal488/PCI Windows 2000 Driver Release 1.0.6  Feb 21 2001
Personal488 Driver Library Release 2.6.6 Mar  5 2001
```

Step 3

Press the ***OpenName*** button.

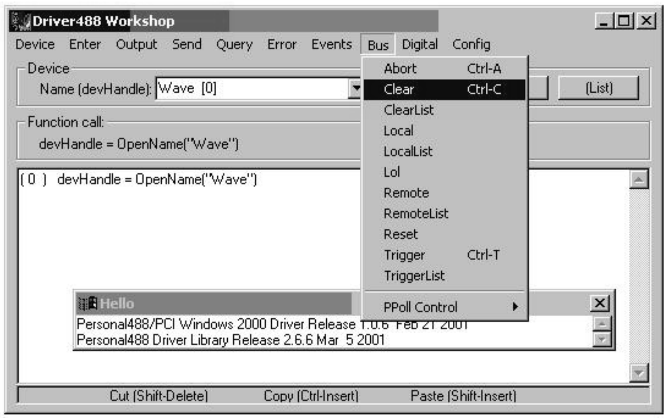
A Hello window will open and display driver revisions.

Note: If an error occurred and the Hello window did not appear, then the installation failed and Wintest did not access the drivers. If this occurs run the installation program found on the IEEE 488 Software CD.



Step 4

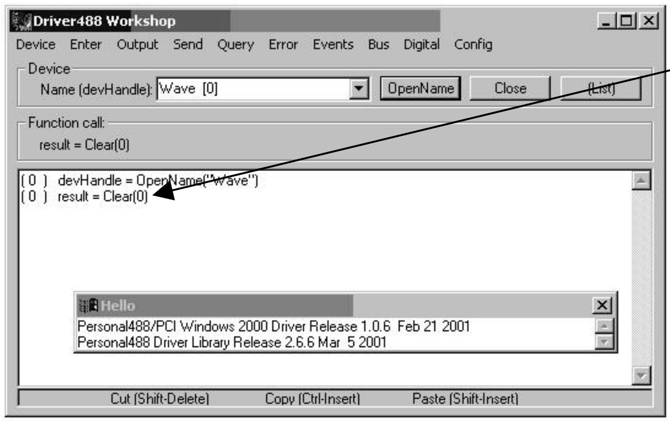
Connect an IEEE 488 instrument to the Personal488 interface.



```
Wave  [0]
devHandle = OpenName("Wave")
( 0 )  devHandle = OpenName("Wave")
Personal488/PCI Windows 2000 Driver Release 1.0.6  Feb 21 2001
Personal488 Driver Library Release 2.6.6 Mar  5 2001
```

Step 5

From the ***Bus*** menu select ***Clear***.


Driver488 Workshop
Device Enter Output Send Query Error Events Bus Digital Config
Device
Name (devHandle):
OpenName
Close
(List)
Function call:
Hello
Cut (Shift-Delete) Copy (Ctrl-Insert) Paste (Shift-Insert)

```
Wave  [0]
result = Clear(0)
( 0 )  devHandle = OpenName("Wave")
( 0 )  result = Clear(0)
Personal488/PCI Windows 2000 Driver Release 1.0.6  Feb 21 2001
Personal488 Driver Library Release 2.6.6 Mar  5 2001
```

Step 6

Verify that ***(0)result = Clear(0)*** appears in the function display, as shown.

If Wintest hangs for a brief period and then returns ***(-1)result = Clear(0)*** then the interface test has failed.

The ***Clear*** command verifies two levels of communication. *First*, it verifies that your instrument will accept a series of commands from the IEEE 488 Controller. *Second*, it verifies that Driver488 was able to receive confirmation of command execution (IRQ) from the interface.

The ***Clear*** command is an ideal test command since it writes only low-level commands to the IEEE 488 bus. No data is transferred during the execution of the ***Clear*** command. Because all standard IEEE 488 devices read and evaluate low-level commands; the ***Clear*** command test avoids issues such as the instrument's IEEE 488 address, device dependent commands and EOS characters that would otherwise be necessary to know in advance.

How do I communicate with my instrument?

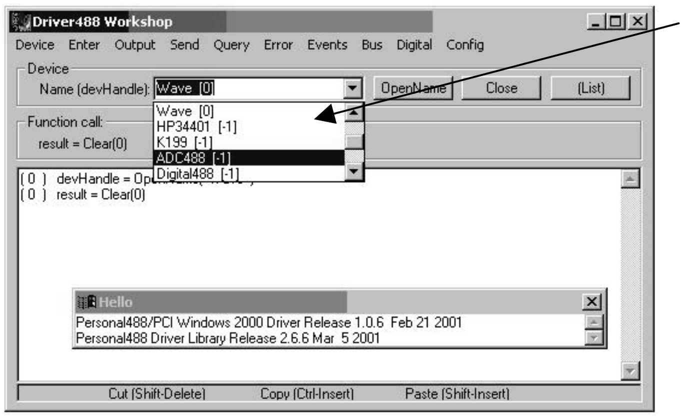
With the exception of transferring binary data or uploading large amounts of data to devices, Wintest can be used to thoroughly exercise IEEE 488 devices.

Start Wintest and click the ***devHandle*** field, which lists your pre-configured devices. For more information regarding pre-configured devices, refer to the section ***IEEE 488 Configuration Utility***. If no changes were made to the factory default, the table should include a device called '***Wave'***. Select the device ***'Wave'***, then click the ***OpenName*** button. The number [–1] in brackets will change to a positive number (greater than –1) indicating that the device handle has been assigned. This positive number is the device's handle.

What is a device handle?

In its simplest form, a handle is merely a reference or pointer to a specific device on the IEEE 488 bus. One might ask why not use the IEEE 488 address instead of handles? Drivers of the past did indeed work this way, however, when issues such as devices with different EOS terminators or time-out setting arose, control became difficult. Hence, handles are method of conveying not just an address, but also EOS and time-out information for each device. In addition, it is valid to have multiple handles for one device. For example, a device might use EOS terminators for setup commands, but when data is downloaded, specifically binary, it might be more convenient to have a handle where EOS is set to none.

If you have not already done so, select your device from the ***devHandle*** drop down box and press the ***OpenName*** button. For the sake of discussion, we chose our device called ***Wave***.



```
result = Clear(0)
```

```
( 0 )  devHandle = Op
( 0 )  result = Clear(0)
```

```
Personal488/PCI Windows 2000 Driver Release 1.0.6  Feb 21 2001
Personal488 Driver Library Release 2.6.6 Mar  5 2001
```

The **Name (devHandle)** combo box lists all the available pre-configured devices.

When the **OpenName** button is pressed, the function **OpenName** is actually called. The parameter that **OpenName** accepts is a string that represents a name of a pre-configured device. If the string name does not exist, a handle is not created. For this example, we see that the device ***Wave*** is now open and the function display area shows the actual function call with parameters.

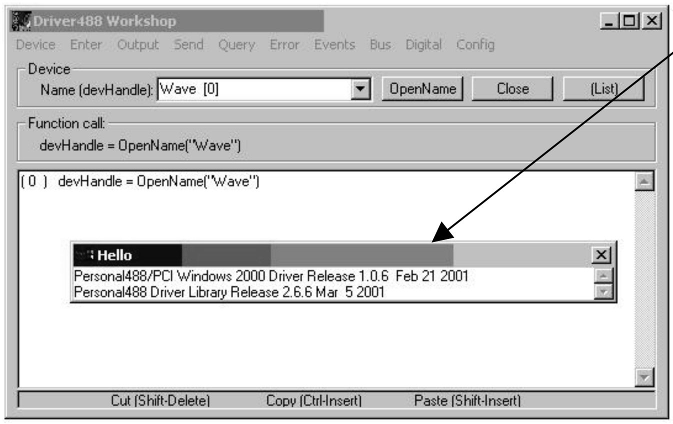
In the function display area, the left-hand number in parenthesis is the value returned by the function call during execution. Generally, if the command is successfully completed, the number is zero (0) and if an error occurred it is minus one (–1). Other functions such as ***Output***, ***Enter*** and ***Spoll*** return more meaningful values such as the number of characters transferred or the serial poll status byte. Refer to the API reference section for more detailed meanings of function return values.

A typical communication sequence consists of

- **opening a device**
- **sending setup commands**
- **reading data**

The following screen shots demonstrate this sequence.

Note that opening a device returns a handle that is used in all subsequent calls. The handle is merely an instrument descriptor containing address, terminator, and time out information.

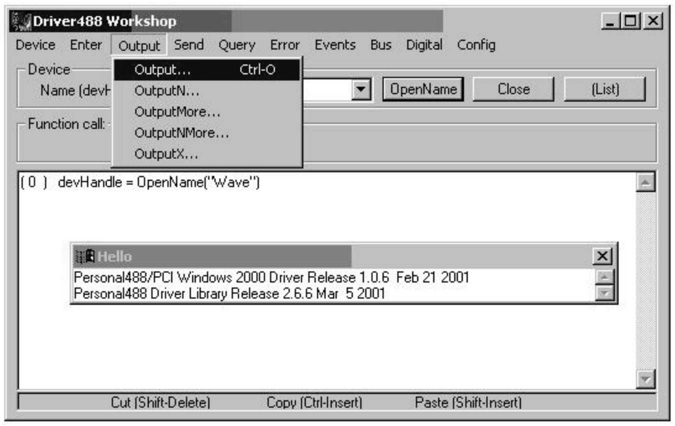

Driver488 Workshop
Device Enter Output Send Query Error Events Bus Digital Config
Device
Name (devHandle): Wave [0]
OpenName Close (List)
Function call:
Hello
Cut (Shift-Delete) Copy (Ctrl-Insert) Paste (Shift-Insert)

```
devHandle = OpenName("Wave")
(0 )  devHandle = OpenName("Wave")
Personal488/PCI Windows 2000 Driver Release 1.0.6  Feb 21 2001
Personal488 Driver Library Release 2.6.6 Mar  5 2001
```

Upon opening the first device, a Hello response window appears.

The response window displays data information.

To write data or device-dependent setup commands to an instrument use the ***Output*** function. The ***Output*** function is a general-purpose write function. However, passing the handle to ***Output*** temporarily converts it to a write command specific to one instrument. When selecting the ***Output*** function, Wintest will automatically use the most current handle.

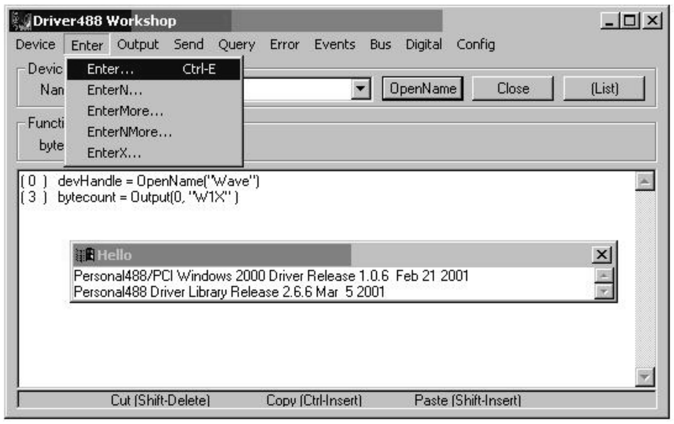


```
(0 )  devHandle = OpenName("Wave")
Personal488/PCI Windows 2000 Driver Release 1.0.6  Feb 21 2001
Personal488 Driver Library Release 2.6.6 Mar  5 2001
```

Select the **`Output`** command. An options panel will open where the only enable option is the data string to write. In this example we will enter ‘W1X’ in the Output Dialog box (not show). ‘W1X’ programs our device to generate a specific wave-form type. Keep in mind that the string of characters ‘W1X’ is a device-dependent command exclusive to our Wave device. You may have an instrument with a different command set.

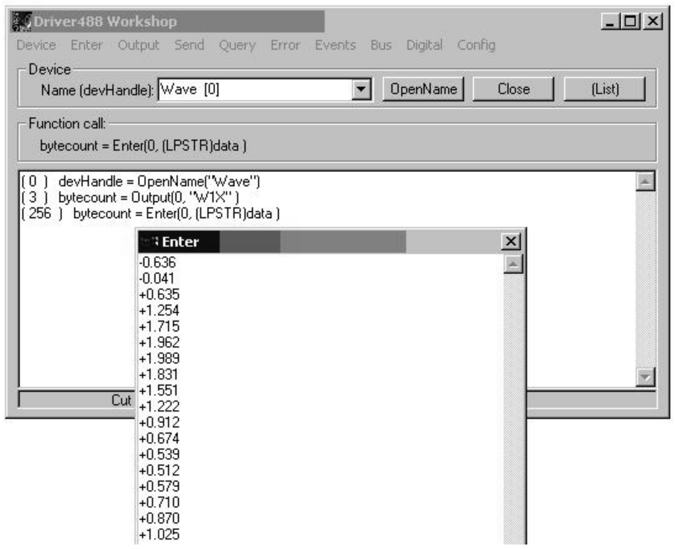
Refer to the API reference for additional information regarding Output commands

To read data from an instrument, use the ***Enter*** functions. The ***Enter*** function is a general-purpose read function. However, passing the handle to ***Enter*** temporarily converts it from general-purpose to a read function specific to one instrument. When selecting the ***Enter*** function, Wintest will automatically use the most current handle.


Driver488 Workshop
Device Enter Output Send Query Error Events Bus Digital Config
Enter... Ctrl-E
EnterN...
EnterMore...
EnterNMore...
EnterX...
Devic
Nan
Functi
byte
OpenName Close (List)
Hello
Cut (Shift-Delete) Copy (Ctrl-Insert) Paste (Shift-Insert)

```
(0 )  devHandle = OpenName("Wave")
(3 )  bytecount = Output(0, "W1X" )
Personal488/PCI Windows 2000 Driver Release 1.0.6  Feb 21 2001
Personal488 Driver Library Release 2.6.6 Mar  5 2001
```

Select a command from the ***Enter*** menu.



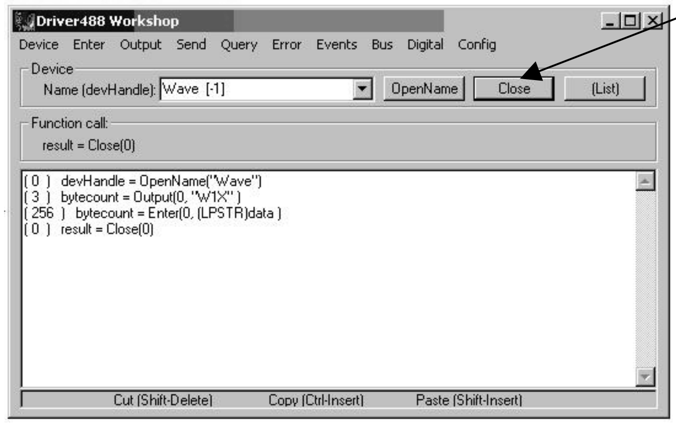
Wave [0]

bytecount = Enter(0, (LPSTR)data)

```
( 0 )  devHandle = OpenName("Wave")
( 3 )  bytecount = Output(0, "W1X" )
( 256 )  bytecount = Enter(0, (LPSTR)data )
```

```
-0.636
-0.041
+0.635
+1.254
+1.715
+1.962
+1.989
+1.831
+1.551
+1.222
+0.912
+0.674
+0.539
+0.512
+0.579
+0.710
+0.870
+1.025
```

For this example, we select the **`Enter`** command. The **`Enter`** command, returns the number of characters actually read. In this example the return is 256.



Wave [-1]

result = Close(0)

```
( 0 )  devHandle = OpenName("Wave")
( 3 )  bytecount = Output(0, "W1X" )
( 256 )  bytecount = Enter(0, (LPSTR)data )
( 0 )  result = Close(0)
```

The **`Close`** command is used to close an open device handle. **`Close`** should be called before ending a program. For the most part, all handles will be forcibly closed when a program terminates. However, in some instances handles can remain active due to the driver remaining in memory after program termination. If a handle remains open, no other programs will be able to access it including the program that left the handle open.

This completes a simple transaction. Your application will no doubt be more complex; however, it will still consist of

- **opening a device name**
- **programming a device**
- **reading data from the device**

In this example we only discussed a small subset of the available API commands. You can use WinTest to explore other API function calls. Refer to the API Reference section of this manual for additional information regarding each command.

Differences Between 32-bit and 16-bit Driver488 Software

General Differences

The following "bulleted items" are the general differences between the 32-bit Driver488 software (for Windows 9x and Windows NT) and the16-bit Driver488 software (for Windows 3.X).

With the 32-bit driver:

- There is no RS-232 serial support.
- The function **`Hello`** now returns two lines of ID: One for the Dynamic Link Library (DLL) and one for the device driver.
- The library function prototypes have changed to reflect standard Windows types.
- The **`include`** file has been renamed to: **`IOTIEEE.H`**

Specific Differences

The following highlight the specific differences in the API command functions, between the 32-bit Driver488 software and the16-bit Driver488 software.

Functions Obsolete for 32-bit Drivers
The parameters that these functions set are now set by a provided Windows Control Panel configuration utility and have therefore been made obsolete for the 32-bit driver.

`ClockFrequency`	**`IOAddress`**
`DmaChannel`	**`LightPen`**
`IntLevel`	**`SysController`**

Functions Supported by 32-bit Drivers
The following functions are supported by 32-bit drivers, but not by 16-bit drivers.

`MakeNewDevice`	**`TermQuery`**
`OnEventVDM` (Console mode applications)	**`TimeOutQuery`**

Functions that are Enhanced in the 32-bit Drivers
These are updated functions:

`ControlLine`	**`KeepDevice`**
`Hello`	

Programming Language Support

Driver488/W95 and Driver488/WNT both provide native language support for Microsoft Visual C++, Visual Basic, Borland C++ and Borland Delphi. The following sections describe support for these 32-bit languages. In some cases, instructions are provided for particular versions of the language.

Although instructions are provided, compiler manufacturers may change their methods, thereby making the instructions obsolete for a particular compiler. This does not mean that the language and version lack support. It does, however, indicate that modifications may be needed as directed by the manufacturer of the compiler.

Microsoft Visual C++

This section is based on use with 32-bit Microsoft Visual C++ V6.0. The procedure may need modified for other versions of Visual C++ . If this is the case, refer to Microsoft documentation for the required changes.

Although the project build procedures may differ, earlier 32-bit versions of Visual C++ (V2.0 or later) should function well.

If using versions earlier than V2.0, e.g., V1.0 or V1.56, support can only be attained by using the 16-Bit Driver488/W95 Compatibility Layer [see section below] since these versions of Visual C++ are 16-bit only. In this case, the Visual C++ support from the Driver488/W31 should be used to develop and build the application.

The **IOTSLPIB.DLL** and corresponding export library **IOTSLPIB.LIB** were developed and built using Microsoft Visual C++. Therefore, all that is needed is to include the **IOTIEEE.H** file into the source code include statements and link to the **IOTSLPIB.LIB** export library file located.

By default, all language support files and examples are located in:

<InstallDirectory>\IEEE488\Programming Language Support\Example Programs\C

To begin your first project, perform the following:

1. Launch Microsoft Visual C++ from Developers Studio.
2. Under the **File** menu select **New** then select **Project.**
3. If using an existing example then select **Console Mode Application** otherwise select a project type which best serves your needs.
4. Follow the project wizard instructions.
5. Under the **Project** menu select **Add to Project** then select **Files**.
6. Add the **IOTIEEE.H**, **IOTERROR.H** , **IOTSLPIB.LIB** files to the project by browsing to the language support described above.
7. If using an existing example then add the example to the project in the same manner. Otherwise use existing or new C++ files by placing **#include IOTIEEE.H** and **#include IOTERROR.H** statements before any references to IEEE 488 functions in the file.
8. Under the **Build** menu select **Build** or **Build All**.

   The project should now be built.
9. Save your project by selecting **File** menu then selecting **Save Workspace**.

Borland C++

This section is based on use with 32-bit Borland C++ V6.0. The procedure may need modified for other versions of Borland C++. If this is the case, refer to Borland C++ documentation for the required changes.

Though the project build procedures may be different, earlier 32-bit versions of Borland C++ (4.0 or later) should function as well as version 6.0.

If using versions earlier than V4.0, such as V3.11, support *can only be attained* by using the 16-Bit Driver488/W95 Compatibility Layer (see section below). This is because these versions of Borland C++ are only 16-bit. In this case, the Borland C++ support from the Driver488/W31 should be used to develop and build the application.

The **IOTSLPIB.DLL** has been developed and built using Microsoft Visual C++. However, a Borland C++ compatible export library **IOTSLPIB.LIB** is available.

Borland C++ **IOTSLPIB.LIB** and other language support files and examples are located in:

<InstallDirectory>\IEEE488\Programming Language Support\Example Programs\C

To begin your first project, perform the following:

1. Launch Borland C++ IDE.
2. Start a new project.
3. Add **IOTSLPIB.LIB** to the project.
4. Add an existing example program or create a new .cpp file.
5. If creating a new .cpp file place include statements for **IOTIEEE.H** and **IOTERROR.H** before any references to the IEEE 488 functions.
6. Set the “Pre-compiled header” option to “None.”
7. Set the “Treat enum types as ints” option.
8. Save the project.

As of the writing of this manual these options are located on the Compiler tab on the Program Options dialog in the Borland C++ IDE. However, these settings are subject to change by the compiler manufacturer.

Microsoft Visual Basic

This section is based on 32-bit Microsoft Visual Basic V4.0, V5.0 and V6.0. The procedure may need modified for other versions of Visual Basic. If this is the case, refer to Microsoft documentation for the required changes.

If using earlier versions, such as V1.0, V2.0 or V3.0 then support can only be attained by using the 16-Bit Driver488/W95 Compatibility Layer (see section below) since these versions of Visual Basic are 16-bit only. In this case, the Visual Basic support from the Driver488/W31 should be used to develop and build the application.

Visual Basic language support files and examples are located in:

<InstallDirectory>\IEEE488\Programming Language Support\Example Programs\VB

To begin your first project perform the following:

To use Visual BasicV5.0 - V6.0 with a supplied example:

1. Launch Visual Basic
2. From the "New Project" dialog box select the "Existing" tab.
3. To use Visual Basic support from an existing example then navigate to the Visual Basic project examples and select the desired project.

If the default settings were accepted during installation the project examples will be located in

C:\Program Files\IEEE488\Programming Language Support\Example Programs\VB

The Visual Basic examples were created using a sub 6.0 version. This results in a message during the load operation that simply states the file was created using a previous version of VB. If the file is saved it will be saved in the 6.0 format.

To use Visual Basic V4.0 with a supplied example:

1. Launch Visual Basic.
2. From the *File* menu select *Open Project.*
3. Add the Visual Basic header file by navigating to the IEEE 488 Visual Basic project folder. If the default settings were accepted during installation the project examples will be located at

   C:\Program Files\IEEE488\Programming Language Support\Example Programs\VB.
4. Add **IOTIEEE.BAS**.
5. Navigate to the IEEE 488 Visual Basic project folder and select the desired example.

To add IEEE 488 support to a new or existing Visual Basic project:

1. Launch Visual Basic.
2. Open a new project.
3. Include the **IOTIEEE.BAS** file by selecting *Project ⇒ Add Module*.
4. Select the "**Existing**" tab and navigate to the IEEE 488 Visual Basic project folder.
   If the default settings were accepted during installation the folder should be located at

   C:\Program Files\IEEE488\Programming Language Support\Example Programs\VB.

Borland Delphi

This section is based on 32-bit Borland Delphi V4.0. The procedure may need modified for other versions of Borland Delphi. If this is the case, refer to Borland documentation for the required changes.

Although the project build procedures may differ, earlier 32-bit versions of Borland Delphi (V2.0 or later) should function well.

If using versions earlier than V2.0, such as V1.0 or V1.56 *support can only be attained* by using the 16-Bit Driver488/W95 Compatibility Layer (see section below). This is because these versions of Visual C++ are only 16-bit. In this case, the Visual C++ support from the Driver488/W31 should be used to develop and build the application.

The location for the Borland Delphi language support files (32-bit) and examples is

<InstallDirectory>\IEEE488\Programming Language Support\Example Programs\Delphi.

To use Delphi V4.0 with a supplied example:

1. Launch Delphi.
2. If starting from one of the supplied examples, select ***Open Project*** from the ***File*** menu.
3. If using Delphi support from an existing example, navigate to the Delphi project examples.

If the default settings were accepted during installation the project examples will be located in

C:\ProgramFiles \IEEE488\Programming Language Support\Example Programs\Delphi.

4. Select the desired example.

To add IEEE 488 support to a new or existing Delphi project:

1. Launch Delphi.
2. Include the **IOTIEEE.PAS** file by selecting *Project ⇒ Add to Project*.
3. Navigate to the IEEE 488 Delphi project folder.

If the default settings were accepted during installation the project examples will be located in

C:\Program Files\IEEE488\Programming Language Support\Example Programs\Delphi.

Support for Other Languages

Any language capable of dynamically linking to a DLL may be used with this product. Although the **IOTSLPIB.DLL** was built using Microsoft Visual C++, the DLL may be dynamically linked to any application that can dynamically link to a DLL.

The same is true for languages supported under Windows3.1, but not included with the Driver488/WIN product. If the desired language is a 16-bit language that can dynamically link to a 16-bit DLL, then the 16-Bit Driver488/W95 Compatibility Layer (**DRVR488.DLL**) may be used, as discussed shortly.

If using a 32-bit C language, other than those mentioned in this chapter, it may be possible to statically link to the **IOTSLPIB.DLL**. However, you may not be able to link using the **IOTSLPIB.LIB** in its original Microsoft Visual C++ format. Many C language compilers have import facilities that allow importing of Microsoft library (**.LIB**) files into a library file format native to the particular compiler. Consult your compiler's documentation for further information on importing Microsoft Visual C++ library files.

16-Bit Driver488/W95 Compatibility Layer

Unlike Driver488/WNT, Driver488/W95 supports backward compatibility for applications written in the 16-bit environment of the Driver488/W31 (formerly named Driver488/WIN) product. Support is provided through a Dynamic Link Library [`DRVR488.DLL`] and through various language-specific header files that will allow recompilation of 16-bit applications.

Note! If Driver488/W31 has been previously installed on your Windows/9x system, the setup program will automatically replace the 16-bit version DLL in the WINDOWS\SYSTEM directory.

Notes

API Reference 6

This chapter contains the API command reference for Driver488/W95 and Driver488/WNT, *using the C language*. The following commands are presented in alphabetical order for ease of use.

Abort

Syntax	**`INT WINAPI Abort(DevHandleT devHandle);`**
	`devHandle` refers to either an IEEE 488 hardware interface or an external device. If **`devHandle`** refers to an external device, the **`Abort`** command will act on the hardware interface to which the external device is attached.
Returns	**`-1`** if error
Bus States	**`IFC, *IFC`** (if **`SC`**)
	`ATN•MTA` (if **`*SC•CA`**)
Example	**`errorflag = Abort(ieee);`**
See Also	**`MyTalkAddr, Talk, UnTalk`**

As the System Controller (**`SC`**), whether Driver488 is the Active Controller or not, the **`Abort`** command causes the Interface Clear (**`IFC`**) bus management line to be asserted for at least 100 microseconds. By asserting **`IFC`**, Driver488 regains control of the bus even if one of the devices has locked it up during a data transfer. Asserting **`IFC`** also makes Driver488 the Active Controller. If a Non System Controller was the Active Controller, it is forced to relinquish control to Driver488. **`Abort`** forces all IEEE 488 device interfaces into a quiescent state.

If Driver488 is a Non System Controller in the Active Controller state (**`*SC•CA`**), it asserts Attention (**`ATN`**), which stops any bus transactions, and then sends its My Talk Address (**`MTA`**) to "Untalk" any other Talkers on the bus. It does not (and cannot) assert **`IFC`**.

Arm

Syntax	**`INT WINAPI Arm(DevHandleT devHandle, ArmCondT condition);`**
	`devHandle` refers to either an IEEE 488 hardware interface or an external device. If **`devHandle`** refers to an external device, the **`Arm`** command acts on the hardware interface to which the external device is attached.
	`condition` is one of the following: **`acSRQ`**, **`acDigMatch`**
Returns	**`-1`** if **`DevHandleT`** is an illegal device or interface
	otherwise, the current state of the event trigger flag
Bus States	None
Example	**`errorflag = Arm(ieee, acSRQ);`**
See Also	**`Disarm, OnEvent, DigArmSetup, DigSetup`**

The **`Arm`** command allows Driver488 to signal to the user-specified function when one or more of the specified conditions occurs. **`Arm`** sets a flag for each implementation of the conditions that are user-indicated. **`Arm`** conditions may be combined using the bitwise **`OR`** operator.

The following **`Arm`** conditions are supported:

Condition	Description
acSRQ	The Service Request bus line is asserted.
acDigMatch	The digital port match condition has occurred. See notes.

Notes:

(1) The acDigMatch event is only available with the Personal488/ATpnp and the Personal488/PCI.

(2) The use of the acDigMatch event requires configuration of the digital port via DigSetup and setting a match value via DigArmSetup.

AutoRemote

Syntax	`INT WINAPI AutoRemote(DevHandleT devHandle, BOOL flag);`
	`devHandle` refers to either an IEEE 488 hardware interface or an external device. If **`devHandle`** refers to an external device, the **`AutoRemote`** command acts on the hardware interface to which the external device is attached.
	`flag` may be either **`OFF`** or **`ON`**
Returns	**`-1`** if **`DevHandleT`** is an illegal device or interface
	otherwise, the previous state is returned
Bus States	None
Example	`errorcode = AutoRemote(ieee,ON);`
See Also	`Local, Remote, EnterX, OutputX`

The **`AutoRemote`** command enables or disables the automatic assertion of the Remote Enable (**`REN`**) line by **`Output`**. When **`AutoRemote`** is enabled, **`Output`** automatically asserts **`REN`** before transferring any data. When **`AutoRemote`** is disabled, there is no change to the **`REN`** line. **`AutoRemote`** is on by default.

BusAddress

Syntax	**`INT WINAPI BusAddress (DevHandleT devHandle, BYTE primary, BYTE secondary);`**
	`devHandle` refers to either an IEEE 488 hardware interface or an external device.
	`primary` is the IEEE 488 bus primary address of the specified device.
	`secondary` is the IEEE 488 bus secondary address of the specified device. If the specified device is an IEEE 488 hardware interface, this value must be **`-1`** since there are no secondary addresses for the IEEE 488 hardware interface. For no secondary address, a **`-1`** must be specified.
Returns	**`-1`** if error
Bus States	None
Example	**`errorcode = BusAddress(dmm,14,0);`**
See Also	**`MakeDevice`**

The **`BusAddress`** command sets the IEEE 488 bus address of the IEEE 488 hardware interface or an external device. Every IEEE 488 bus device has an address that must be unique within any single IEEE 488 bus system. The default IEEE 488 bus address for Driver488 is **`21`**, but this may be changed if it conflicts with some other device.

CheckListener

Syntax	`INT WINAPI CheckListener(DevHandleT devHandle, BYTE primary, BYTE secondary);`
	`devHandle` refers to either an IEEE 488 hardware interface or an external device. If `devHandle` refers to an external device, the `CheckListener` command acts on the hardware interface to which the external device is attached.
	`primary` is the primary bus address to check for a Listener (`00` to `30`)
	`secondary` is the secondary bus address to check for a Listener (`00` to `31`). For no secondary address, a `-1` must be specified
Returns	`-1` if error
	otherwise it returns a `1` if a listener was found at the specified address, or a `0` if a listener was not found at the specified address.
Bus States	`ATN•UNL, LAG,` (check for `NDAC` asserted)

Example	`result = CheckListener(ieee,15,4);`
	`if (result == 1)`
	`{`
	`printf("Device found at specified address.\n");`
	`}`
	`if (result == 0)`
	`{`
	`printf("Device not found at specified address.\n");`
	`}`

See Also	`FindListener, BusAddress`

The `CheckListener` command checks for the existence of a device on the IEEE 488 bus at the specified address.

Clear

Syntax	`INT WINAPI Clear(DevHandleT devHandle);`	
	`devHandle` refers to either an IEEE 488 hardware interface or an external device. If `devHandle` refers to a hardware interface, then a Device Clear (`DCL`) is sent. If `devHandle` refers to an external device, a Selected Device Clear (`SDC`) is sent.	
Returns	`-1` if error	
Bus States	`ATN•DCL` (all devices)	
	`ATN•UNL, MTA, LAG, SDC` (selected device)	
Examples	`errorcode = Clear(ieee);`	Sends the Device Clear (`DCL`) command to the IEEE interface board.
	`errorcode = Clear(wave);`	Sends the Selected Device Clear (`SDC`) command to the WAVE device.
	`errorcode = Clear(dmm);`	Sends the Selected Device Clear (`SDC`) command to the DMM device.
See Also	`Reset, ClearList`	

The `Clear` command causes the Device Clear (`DCL`) bus command to be issued to an interface or a Selected Device Clear (`SDC`) command to be issued to an external device. IEEE 488 bus devices that receive a Device Clear or Selected Device Clear command normally reset to their power-on state.

ClearList

Syntax	**`INT WINAPI ClearList(DevHandlePT devHandleList);`**
	`devHandleList` is a pointer to a list of device handles that refer to external devices. If a hardware interface is in the list, **`DCL`** is sent instead of **`SDC`**.
Returns	**`-1`** if error
Bus States	**`ATN•DCL`** (all devices)
	`ATN•UNL, MTA, LAG, SDC` (selected device)

Example	`deviceList[0] = wave;`
	`deviceList[1] = scope;`
	`deviceList[2] = dmm;`
	`deviceList[3] = NODEVICE;`
	`errorcode =`
	`ClearList(deviceList);` — Sends the Selected Device Clear (**`SDC`**) command to a list of devices.

See Also	**`Clear, Reset`**

The **`ClearList`** command causes the Selected Device Clear (**`SDC`**) command to be issued to a list of external devices. IEEE 488 bus devices that receive a Selected Device Clear command normally reset to their power-on state.

Close

Syntax	**`INT WINAPI Close(DevHandleT devHandle);`**
	`devHandle` refers to either an IEEE 488 interface or an external device.
Returns	**`-1`** if error
Bus States	Completion of any pending I/O activities
Example	**`errorcode = Close(wave);`**
See Also	**`OpenName, MakeDevice, Wait`**

The **`Close`** command waits for I/O to complete, flushes any buffers associated with the device that is being closed, and then invalidates the handle associated with the device.

ControlLine

Syntax	`INT WINAPI ControlLine(DevHandleT devHandle);`
	`ControlLine` returns a bit mapped number.
	`devHandle` refers to the I/O adapter. If `devHandle` refers to an external device, the `ControlLine` command acts on the hardware interface to which the external device is attached.
Response	`-1` if error
	otherwise, a bit map of the current state of the IEEE 488 interface. Under 32-bit Driver488 software, serial interfaces are no longer supported.
Bus States	None
Example	`result = ControlLine(ieee);` `printf("The response is %X\n",result);`
See Also	`TimeOut`

The `ControlLine` command may be used only on IEEE 488 devices. Under 32-bit Driver488 software, serial interfaces are no longer supported. This command returns the status of the IEEE 488 bus control lines as an 8-bit unsigned value (bits 2 and 1 are reserved for future use), as indicated below.

Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
EOI	SRQ	NRFD	NDAC	DAV	ATN	0	0

DigArm

AT488pnp and PCI488 Only !	
Syntax	`INT WINAPI DigArm(DevHandleT devHandle, BOOL bArm);`
	`devHandle` refers to an interface handle.
	`bArm` refers to a value that arms or disarms event generation. `TRUE` = Arm, `FALSE` = Disarm.
Returns	`-1` if neither nibble is set for input, or other error
Bus States	None
Example	`DigArm(devHandle, TRUE);` Arms digital input event generation.
See Also	`DigArmSetup, DigSetup, OnDigEvent, OnDigEventVDM`

The `DigArm` command arms or disarms the event-generation due to a digital I/O port match condition. The caller should configure the digital I/O port, the event-callback mechanism, and the match condition prior to arming the event generation. The following code snippet illustrates this sequence:

```
DigSetup(devHandle, FALSE, FALSE);    // Configure both nibbles for input.
OnDigEventVDM(devHandle, MyFunc, 0);  // On event, call function MyFunc.
DigArmSetup(devHandle, 0x0A5);        // Trigger when inputs equals 0xA5.
DigArm(devHandle, TRUE);              // Enable event generation.
```

Event generation is automatically disarmed when an event is triggered. The event generation configuration, however, remains intact, so event generation can be re-armed just by calling `DigArm`. The other steps shown in the above code snippet do not need to be repeated unless the event configuration is to be changed.

Event generation may be disarmed (`bArm = FALSE`) at any time.

Notes:

(1) This function does not configure the digital I/O port for input. The caller must use `DigSetup` to configure the port for input before performing arming event generation. If neither nibble is configured for input the function returns `-1` and sets the error code to `IOT_BAD_VALUE2`.

(2) Event generation may be re-armed from within the event handler to provide continuous detection of match condition events. However, this is not guaranteed to catch every event if the digital input values are rapidly changing.

(3) Any digital I/O port bits configured for output are treated as “don’t care” bits for the purposes of event generation. In other words, it is valid to arm an event when only one nibble of the port is configured for input. In this case, the other nibble is ignored when detecting the match condition.

DigArmSetup

AT488pnp and PCI488 Only !	
Syntax	**`INT WINAPI DigArmSetup(DevHandleT devHandle, BYTE byMatchValue);`**
	`devHandle` refers to an interface handle.
	`byMatchValue` refers to a value that is compared against the digital I/O inputs
Returns	**`-1`** if error
Bus States	None
Example	**`DigArmSetup(devHandle, 0xA5);`** Sets the match value to **`0xA5`**.
See Also	**`DigArm, DigSetup`**

The **`DigArmSetup`** command sets the match condition value. This value will be compared against the digital I/O port inputs to detect when an event occurs. The event must be armed (via **`DigArm`**) for event notification to take place.

The comparison operation depends on the current digital port configuration. If both nibbles are configured for input, then the match value is compared to the entire byte value of the digital port. If only one of the nibbles is configured for input, then the value is compared against just that nibble. If no nibbles are configured for input, then the match value is ignored. The **`DigArm`** function will not allow event generation to be armed unless at least one of the nibbles is configured for input.

DigRead

AT488pnp and PCI488 Only !	
Syntax	**`INT WINAPI DigRead(DevHandleT devHandle);`**
	`devHandle` refers to an interface handle.
Returns	**`-1`** if no part of the port is configured for input, or other error
	otherwise, integer between `0` and `255` if the entire digital I/O port is configured for input; or integer between `0` and `15` if only one nibble (either low or high) is configured for input
Bus States	None
Example	**`int i = DigRead(devHandle);`** Returns the current value of the digital I/O port per the current configuration.
See Also	**`DigSetup, DigWrite`**

The **`DigRead`** command reads the current value of the digital IO port per the input/output configuration of the port. If the entire port is configured for input, a value between `0` and `255` is returned. If either the upper or lower nibble is configured for input, and the other for output, a value between `0` and `15` is returned.

Note: This function does not configure the digital I/O port for input. The caller must use **`DigSetup`** to configure the port for input before performing any reads. If neither nibble is configured for input the function returns **`-1`** and sets the error code to **`IOT_BAD_VALUE2`**.

DigSetup

AT488pnp and PCI488 Only !	
Syntax	`INT WINAPI DigSetup(DevHandleT devHandle, BOOL bLowOut, BOOL bHighOut);`
	`devHandle` refers to an interface handle.
	`bLowOut` refers to the lower nibble setup. `TRUE` = output, `FALSE` = input.
	`bHighOut` refers to the upper nibble setup. `TRUE` = output, `FALSE` = input.
Returns	`-1` if error
Bus States	None
Examples	`DigSetup(devHandle, TRUE , TRUE);` All 8 bits output.
	`DigSetup(devHandle, FALSE, TRUE);` Lower 4 bits input, upper 4 output.
	`DigSetup(devHandle, TRUE , FALSE);` Lower 4 bits output, upper 4 input.
	`DigSetup(devHandle, FALSE, FALSE);` All 8 bits input.
See Also	`DigRead, DigWrite`

The **`DigSetup`** command configures the digital I/O port for input and output on a per-nibble basis. Each of the two nibbles can be set for input or output. All combinations are supported. Once **`DigSetup`** is called, the configuration of the digital I/O port does not change until the next call to **`DigSetup`**. The port may be read and written many times without affecting the port setup.

Note: The digital I/O port must be configured every time the driver is opened. The configuration is not stored between sessions.

DigWrite

AT488pnp and PCI488 Only !	
Syntax	`INT WINAPI DigWrite(DevHandleT devHandle, BYTE byDigData);`
	`devHandle` refers to an interface handle.
	`byDigData` refers to a value to write to the digital output port, where the integer range is between `0` and `255` if the entire digital I/O port is configured for output, or between `0` and `15` if only one nibble (either low or high) is configured for output.
Returns	`-1` if no part of the digital I/O port is configured for output.
Bus States	None
Example	`DigRead(devHandle, 0x0A);` Writes the given value to the digital I/O port per the current configuration.
See Also	`DigSetup, DigRead`

The **`DigWrite`** command writes the given value to the digital I/O port per the input/output configuration of the port. If the entire port is configured for output, then the data value with a range from `0` to `255` is written to the port. If either the upper or lower nibble is configured for input, and the other for output, then the data value is truncated to the range from `0` to `15` and it is written to the appropriate nibble per the current configuration.

Notes:

(1) This function does not configure the digital I/O port for output. The caller must use **`DigSetup`** to configure the port before performing any reads or writes. If neither nibble is configured for output the function returns `-1` and sets the error code to **`IOT_BAD_VALUE2`**.

(2) Outputs do not persist after an interface is closed. At that time, all digital I/O lines are configured for input.

Disarm

Syntax	**`INT WINAPI Disarm(DevHandleT devHandle, ArmCondT condition);`**
	`devHandle` refers to either an IEEE 488 interface or an external device. If **`devHandle`** refers to an external device, then the **`Disarm`** command acts on the hardware interface to which the external device is attached.
	`condition` specifies which of the conditions are no longer to be monitored. If condition is **`0`**, then all conditions are **`Disarm`**ed.
Returns	**`-1`** if error
	otherwise, the current bit map of the event condition mask.
Bus States	None
Examples	**`errorcode=Disarm(ieee,acTalk│acListen│acChange);`**
	`errorcode=Disarm(ieee,0);`
See Also	**`Arm, OnEvent`**

The **`Disarm`** command prevents Driver488 from invoking an event handler and interrupting the PC, even when the specified condition occurs. Your program can still check for the condition by using the **`Status`** command. If the **`Disarm`** command is invoked without specifying any conditions, then all conditions are disabled. The **`Arm`** command may be used to re-enable interrupt detection.

EnterX

Syntax	`LONG WINAPI EnterX(DevHandleT devHandle, LPBYTE data,DWORD count,BOOL forceAddr,TermT*term,BOOL reserved,LPDWORD compStat);`
	`devHandle` refers to either an IEEE 488 interface or an external device.
	`data` is a pointer to the buffer into which the data is read.
	`count` is the number of characters to read.
	`forceAddr` is used to specify whether the addressing control bytes are to be issued for each **`EnterX`** command.
	`term` is a pointer to a terminator structure that is used to set up the input terminators. If **`term`** is set to **`0`**, the default terminator is used.
	`reserved` - this value is ignored.
	`compStat` is a pointer to an integer containing completion status information.
Returns	**`-1`** if error
	otherwise, the actual count of bytes transferred. The memory buffer pointed to by the data parameter is filled in with the information read from the device. Note that the actual count does not include terminating characters if term characters are specified by the term in function. In addition, term characters are not returned but are discarded.
Bus States	With interface handle: **`*ATN, data`**
	With external device handle: **`ATN•UNL, MLA, TAG, *ATN, data`**

Example	`term.EOI = TRUE;`
	`term.nChar = 1;`
	`term.EightBits = TRUE;`
	`term.termChar[0] = '\r';`
	`bytecount=EnterX(timer,data,1024,0,&term,1,&stat);`

See Also	**`OutputX, Term, Buffered`**

The **`EnterX`** command reads data from the I/O adapter. If an external device is specified, then Driver488 is addressed to Listen, and that device is addressed to Talk. If an interface is specified, then Driver488 must already be configured to receive data and the external device must be configured to Talk, either as a result of an immediately preceding **`EnterX`** command or as a result of one of the **`Send`** commands. **`EnterX`** terminates reception on either the specified count of bytes transferred, or the specified or default terminator being detected. Terminator characters, if any, are stripped from the received data before the **`EnterX`** command returns to the calling application.

The **`forceAddr`** flag is used to specify whether the addressing control bytes are to be issued for each **`EnterX`** command. If the device handle refers to an I/O adapter, then **`forceAddr`** has no effect and command bytes are not sent. For an external device, if **`forceAddr`** is **`TRUE`** then Driver488 always sends the **`UNL`**, **`MLA`**, and **`TAG`** command bytes. If **`forceAddr`** is **`FALSE`**, then Driver488 compares the current device with the previous device that used that interface adapter board for an **`EnterX`** command. If they are the same, then no command bytes are sent. If they are different, then **`EnterX`** acts as if the **`forceAddr`** flag were **`TRUE`** and sends the command bytes. The **`forceAddr`** flag is usually set **`TRUE`** for the first transfer of data from a device, and then set **`FALSE`** for additional transfers from the same block of data from that device.

Additional Enter Functions

Driver488 provides additional **`Enter`** routines that are short-form versions of the **`EnterX`** function. The following **`Enter`** functions are already defined in your header file.

Enter

Syntax	`LONG WINAPI Enter(DevHandleT devHandle, LPBYTE data)`
Remarks	`Enter` is equivalent to the following call to `EnterX`: `EnterX(devHandle,data,sizeof(data),1,0L,0,0L);`

The `Enter` function passes the device handle and a pointer to the data buffer to the `EnterX` function. It determines the size of the data buffer that you provided, and passes that value as the `count` parameter. It specifies `forceAddr` is `TRUE`, causing Driver488 to re-address the device. The default terminators are chosen by specifying a `0` as the `term` parameter. Asynchronous transfer is turned off by sending `0` for the `async` parameter, and the completion status value is ignored by sending `0` for the `compStat` parameter.

EnterN

Syntax	`LONG WINAPI EnterN(DevHandleT devHandle,LPBYTE data,int count)`
Remarks	`EnterN` is equivalent to the following call to `EnterX`: `EnterX(devHandle,data,count,1,0L,0,0L);`

The `EnterN` function passes the device handle, the pointer to the data buffer, and the size of the data buffer to the `EnterX` function. It specifies `forceAddr` is `TRUE`, causing Driver488 to re-address the device. The default terminators are chosen by specifying a `0` pointer as the `term` parameter. Asynchronous transfer is turned off by sending `0` for the `async` parameter, and the completion status value is ignored by sending `0` for the `compStat` parameter.

EnterMore

Syntax	`LONG WINAPI EnterMore(DevHandleT devHandle,LPBYTE data)`
Remarks	`EnterMore` is equivalent to the following call to `EnterX`: `EnterX(devHandle,data,sizeof(data),0,0L,0,0L);`

The `EnterMore` function passes the device handle and the pointer to the data buffer to the `EnterX` function. It determines the size of the data buffer that you provided, and passes that value as the `count` parameter. It specifies `forceAddr` is `FALSE`, therefore Driver488 does not address the device if it is the same device as previously used. The default terminators are chosen by specifying a `0` as the `term` parameter. Asynchronous transfer is turned off by sending `0` for the `async` parameter, and the completion status value is ignored by sending `0` for the `compStat` parameter.

EnterNMore

Syntax	`LONG WINAPI EnterNMore(DevHandleT devHandle,LPBYTE data,int count);`
Remarks	`EnterNMore` is equivalent to the following call to `EnterX`: `EnterX(devHandle,data,count,0,0L,0,0L);`

The `EnterNMore` function passes the device handle, the pointer to the data buffer, and the size of the data buffer to the `EnterX` function. It specifies `forceAddr` is `FALSE`; therefore, Driver488 does not address the device if it is the same device as previously used. The default terminators are chosen by specifying a `0` as the `term` parameter. Asynchronous transfer is turned off by sending `0` for the `async` parameter, and the completion status value is ignored by sending `0` for the `compStat` parameter.

Error

Syntax	`INT WINAPI Error(DevHandleT devHandle, BOOL display);`
	`devHandle` refers to either an IEEE 488 interface or an external device.
	`display` indicates whether the error message display should be **`ON`** or **`OFF`**.
Returns	**`-1`** if error
Bus States	None
Example	`errorcode = Error(ieee, OFF);`
See Also	`OnEvent, GetError, GetErrorList, Status`

The **`Error`** command enables or disables automatic on-screen display of Driver488 error messages. Specifying **`ON`** enables the error message display, while specifying **`OFF`** disables the error message display. **`Error ON`** is the default condition.

FindListeners

Syntax	`INT WINAPI FindListeners(DevHandleT devHandle, BYTE primary, LPWORD listener, DWORD limit);`
	`devHandle` refers to either an IEEE 488 interface or an external device. If **`devHandle`** refers to an external device, then the **`FindListeners`** command acts on the hardware interface to which the external device is attached.
	`primary` is the primary IEEE 488 bus address to check.
	`listener` is a pointer to a list that contains all Listeners found on the specified interface board. You must allocate enough memory to accommodate all of the possible Listeners up to the limit that he specified.
	`limit` is the maximum number of Listeners to be entered into the Listener list.
Returns	**`-1`** if error
	otherwise, the number of Listeners found on the interface
Bus States	**`ATN•MTA, UNL, LAG`**

Example	`WORD listeners[5];`
	`errorcode = FindListeners(ieee,10,listeners,5);`

See Also	**`CheckListener, BusAddress, Status`**

The **`FindListeners`** command finds all of the devices configured to Listen at the specified primary address on the IEEE 488 bus. The command first identifies the primary address to check and returns the number of Listeners found and their addresses. Then, it fills the user-supplied array with the addresses of the Listeners found. The number of Listeners found is the value returned by the function. The returned values include the secondary address in the upper byte, and the primary address in the lower byte. If there is no secondary address, then the upper byte is set to **`255`**; hence, a device with only a primary address of **`16`** and no secondary address is represented as **`0xFF10`** or **`-240`** decimal.

GetError

Syntax	**`ErrorCodeT WINAPI GetError(DevHandleT devHandle, LPSTR errText);`**
	`devHandle` refers to either the IEEE 488 interface or the external device that has the associated error.
	`errText` is the string that will contain the error message. If **`errText`** is non-null, the string must contain at least 247 bytes.
Returns	`-1` if error
	otherwise, it returns the error code number associated with the error for the specified device.
Bus States	None

Example	**`errnum = GetError(ieee,errText);`**
	`printf("Error number:%d;%s \n"errnum,errText);`

See Also	**`Error, GetErrorList, Status`**

The **`GetError`** command is user-called after another function returns an error indication. The device handle sent to the function that returned the error indication is sent to **`GetError`** as its **`devHandle`** parameter. **`GetError`** finds the error associated with that device and returns the error code associated with that error. If a non-null error text pointer is passed, **`GetError`** also fills in up to 247 bytes in the string. The application must ensure that sufficient space is available.

GetErrorList

Syntax	`ErrorCodeT WINAPI GetErrorList(DevHandlePT devHandleList, LPSTR errText, DevHandlePT errHandle);`
	`DevHandleList` is a pointer to a list of external devices that was returned from a function, due to an error associated with one of the external devices in the list.
	`errText` is the text string that contains the error message. You must ensure that the string length is at least 247 bytes.
	`errHandle` is a pointer to the device handle that caused the error.
Returns	**`-1`** if error
	otherwise, it returns the error number associated with the given list of devices.
Bus States	None

Example	`char errText[329];`
	`int errHandle;`
	`int errnum;`
	`result = ClearList(list);`
	`if (result == -1) {`
	`     errnum=GetErrorList(list,errText,&errHandle);`
	`     printf("Error %d;%s,at handle %d\n", errnum, errText,`
	`     errHandle);`
	`}`

See Also	**`Error, GetError, Status`**

The **`GetErrorList`** command is user-called, after another function identifying a list of device handles, returns an error indication. The device handle list sent to the function that returned the error indication, is sent to **`GetErrorList`**. **`GetErrorList`** finds the device that returned the error indication, returning the handle through **`errHandle`**, and returns the error code associated with that error. If a non-null error text pointer is passed, **`GetError`** also fills in up to 247 bytes in the string. The application must ensure that sufficient space is available.

Hello

Syntax	`INT WINAPI Hello(DevHandleT devHandle, LPSTR message);`
	`devHandle` refers to either an IEEE 488 interface or an external device. If **`devHandle`** refers to an external device, the **`Hello`** command acts on the hardware interface to which the external device is attached.
	`message` is a character pointer that contains the returned message, which is the version of the Dynamic Link Library (DLL) and the version of the device driver.
Returns	**`-1`** if error
	otherwise, the length in bytes of the message string. The returned byte count will never exceed 247 bytes.
Bus States	None

Example	`char message[247];`
	`result = Hello(ieee,message);`
	`printf("%s\n",message);`

See Also	**`Status, OpenName, GetError`**

The **`Hello`** command is used to verify communication with Driver488, and to read the software revision number. If a non-null string pointer is passed, **`Hello`** fills in up to 247 bytes in the string. The application must ensure that sufficient space is available. When the command is sent, Driver488 returns a string similar to the following:

Driver488 Revision X.X (C)199X ...

where **`X`** is the appropriate revision or year number.

KeepDevice

Syntax	`INT WINAPI KeepDevice(DevHandleT devHandle);`
	`devHandle` refers to an external device.
Returns	`-1` if error
Bus States	None
Example	`errorcode = KeepDevice(scope);`
See Also	`MakeDevice, MakeNewDevice, RemoveDevice, OpenName`

Note: `KeepDevice` will update an existing device or will create a new device in the Registry. This update feature is new and useful. For example, if you wish to change the bus address of the device and make it a permanent change.

The `KeepDevice` command changes the indicated temporary Driver488 device to a permanent device, visible to all applications. Permanent Driver488 devices are not removed when Driver488 is closed. Driver488 devices are created by `MakeDevice` and are initially temporary. Unless `KeepDevice` is used, all temporary Driver488 devices are forgotten when Driver488 is closed. The only way to remove the permanent device once it has been made permanent by the `KeepDevice` command, is to use the `RemoveDevice` command.

Listen

Syntax	`INT WINAPI Listen(DevHandleT devHandle, BYTE primary, BYTE secondary);`
	`devHandle` refers to either an IEEE 488 interface or an external device. If **`devHandle`** refers to an external device, the **`Listen`** command acts on the associated interface.
	`primary` and **`secondary`** specify the primary and secondary addresses of the device which is to be addressed to listen.
Returns	**`-1`** if error
Bus States	**`ATN, LAG`**
Example	`errorcode = Listen (ieee, 12, -1);`
See Also	**`Talk, SendCmd, SendData, SendEoi, FindListener`**

The **`Listen`** command addresses an external device to Listen.

`Local`

Syntax	`INT WINAPI Local(DevHandleT devHandle);`	
	`devHandle` refers to either an IEEE 488 interface or an external device.	
Returns	`-1` if error	
Bus States	`*REN`	
Examples	`errorcode = Local(ieee);`	To unassert the Remote Enable (**REN**) line, the IEEE 488 interface is specified.
	`errorcode = Local(wave);`	To send the Go To Local (**GTL**) command, an external device is specified.
See Also	`LocalList, Remote, AutoRemote`	

In the System Controller mode, the **`Local`** command issued to an interface device, causes Driver488 to unassert the Remote Enable (**REN**) line. This causes devices on the bus to return to manual operation. A **`Local`** command addressed to an external device, places the device in the local mode via the Go To Local (**GTL**) bus command.

LocalList

Syntax	`INT WINAPI LocalList(DevHandlePT devHandleList);`
	`devHandleList` refers to a pointer to a list of external devices.
Returns	`-1` if error
Bus States	`ATN•UNL, MTA, LAG,GTL`

Example	`deviceList[0] = wave;`
	`deviceList[1] = timer;`
	`deviceList[2] = dmm;`
	`deviceList[3] = NODEVICE;`
	`errorcode = LocalList(deviceList);` — Sends the Go To Local (**GTO**) bus command to a list of external devices.

See Also	`Local, Remote, RemoteList, AutoRemote`

In the System Controller mode, the **LocalList** command issued to an interface device, causes Driver488 to unassert the Remote Enable (**REN**) line. This causes devices on the bus to return to manual operation. A **LocalList** command addressed to an external device, places the device in the local mode via the Go To Local (**GTL**) bus command.

Lol

Syntax	**`INT WINAPI Lol(DevHandleT devHandle);`**
	`devHandle` refers to either an IEEE 488 interface or an external device. If **`devHandle`** refers to an external device, the **`Lol`** command acts on the hardware interface to which the external device is attached.
Returns	**`-1`** if error
Bus States	**`ATN•LLO`**
Example	**`errorcode = Lol(ieee);`**
See Also	**`Local, LocalList, Remote, RemoteList`**

The **`Lol`** command causes Driver488 to issue an IEEE 488 LocalLockout (**`LLO`**) bus command. Bus devices that support this command are thereby inhibited from being controlled manually from their front panels.

MakeDevice

Syntax	`INT WINAPI MakeDevice(DevHandleT devHandle, LPSTR name);`
	`devHandle` refers to an existing external device.
	`name` is the device name of the device that is to be made and takes the configuration of the device given by **`devHandle`**.
Returns	**`-1`** if error
	otherwise, the **`DevHandleT`** of the new device. Note that the new device is an exact copy (except for the name) of the specified device as it currently sets in memory and not in the Registry.
Bus State	None
Example	`dmm = MakeDevice(scope,"DMM");` `BusAddress(dmm,16,-1);` Create a device named **`DMM`**, attached to the same I/O adapter as **`scope`** and set its IEEE 488 bus address to 16.
See Also	**`MakeNewDevice, KeepDevice, RemoveDevice, OpenName, Close`**

The **`MakeDevice`** command creates a new temporary Driver488 device that is an identical copy of an already existing Driver488 external device. The new device is attached to the same I/O adapter of the existing device and has the same bus address, terminators, timeouts, and other characteristics. The newly created device is temporary and is removed when Driver488 is closed. **`KeepDevice`** may be used to make the device permanent. To change the default values assigned to the device, it is necessary to call the appropriate configuration functions such as **`BusAddress`**, **`IOAddress`**, and **`TimeOut`**.

MakeNewDevice

Syntax	`DevHandleT WINAPI MakeNewDevice(LPSTR iName, LPSTR aName,BYTE primary,BYTE secondary,TermPT In,TermPT Out,DWORD timeout);`	
	`devHandle` refers to the new external device.	
	`iName` is the user name of the interface on which the device is to be created.	
	`aName` is the user name of the device.	
	`primary` and `secondary` are the secondary and primary bus addresses to be specified. For no secondary address, a `-1` must be specified.	
	`In` and `Out` are pointers to terminator structures specified to set up the respective input and output terminators of the device.	
	`tOut` is the timeout parameter to be specified.	
Returns	`-1` if error	
	otherwise, the `DevHandleT` of the new device, based on the parameters specified.	
Bus State	None	
Example	`DevHandleT anotherDevice;` `anotherDevice = MakeNewDevice("IEEE0", "Scope",13,-1,NULL, NULL,10000);`	Specifies parameters for: Pointer to the interface, pointer to the device name, `primary` and `secondary` addresses, pointers to the term `In` and `Out` structures, and timeout in milliseconds.
See Also	`MakeDevice, KeepDevice, RemoveDevice, OpenName, Close`	

This is a new function in Driver488/W95 and in Driver488/WNT. This function is similar to the `MakeDevice` function except that `MakeNewDevice` will create a new device based on the parameters specified, instead of simply cloning an existing device.

The `MakeNewDevice` command does not save the parameters of the newly created device in the system registry. To keep the device, it is necessary to call the `KeepDevice` function.

Note: The `MakeNewDevice` command will only create, not save, a new device. Interface descriptions are created and maintained by the configuration utility and the IEEE 488 configuration applet in the Windows Control Panel.

MyListenAddr

Syntax	`INT WINAPI MyListenAddr (DevHandleT devHandle);`
	`devHandle` refers to either an interface or an external device. If `devHandle` refers to an external device, the `MyListenAddr` command acts on the associated interface.
Returns	`-1` if error
Bus States	`ATN, MLA`
Example	`errorcode = MyListenAddr (ieee);`
See Also	`MyTalkAddr, Talk, Listen, SendCmd`

The `MyListenAddr` command addresses the interface to Listen.

MyTalkAddr

Syntax	**`INT WINAPI MyTalkAddr (DevHandleT devHandle);`**
	`devHandle` refers to either an interface or an external device. If **`devHandle`** refers to an external device, the **`MyTalkAddr`** command acts on the associated interface.
Returns	**`-1`** if error
Bus States	**`ATN, MTA`**
Example	**`errorcode = MyTalkAddr (ieee);`**
See Also	**`MyListenAddr, Listen, SendCmd`**

The **`MyTalkAddr`** command addresses the interface to Talk.

OnDigEvent

AT488pnp and PCI488 Only !	**Windows9x Only !**
Syntax	`INT WINAPI OnDigEvent(DevHandleT devHandle, HWND hwnd, OpaqueP lParam);`
	`devHandle` refers to an interface handle.
	`hwnd` is the window handle to receive event notification.
	`lParam` value will be passed in the notification message.
Returns	`-1` if error
Bus States	None
Example	`OnDigEvent(devHandle, TRUE, 0x10L);` Sets the event notification to be via a window message to the specified window handle. The value `0x10` will be passed with the message.
See Also	`DigArm, OnDigEventVDM, OnEvent`

The `OnDigEvent` command sets the handle of a window to receive a notification message when a digital match event is triggered. This function uses the same mechanism as the `OnEvent` command. For details, see the description of `OnEvent`.

Note: This function sets the event generation mechanism to be a window notification message, replacing any previously defined event notification mechanism. Only one event notification mechanism can be used at one time.

OnDigEventVDM

AT488pnp and PCI488 Only !	**Windows9x Only !**	
Syntax	`INT WINAPI OnDigEventVDM(DevHandleT devHandle, DigEventFuncT func, OpaqueP lParam);`	
	`devHandle` refers to an interface handle.	
	`func` is a user-defined function to be called when the digital match event is triggered.	
	`lParam` value will be passed in the notification message.	
Returns	`-1` if error	
Bus States	None	
Example	`OnDigEventVDM(devHandle, MyFunc, 0x10L);`	Sets the event notification to be via a function call to the specified callback function. The value `0x10` will be passed to the function.
See Also	`DigArm, OnDigEventVDM, OnEventVDM`	

The **`OnDigEventVDM`** command sets the address of a "C"-style (**`__stdcall`**) function to be called when a digital match event occurs. This function uses a similar mechanism as the **`OnEventVDM`** command. The prototype of the callback function for **`OnDigEventVDM`** is:

void DigEventFunc(DevHandleT devHandle, LPARAM lParam)

The **`lParam`** value that is passed to **`OnDigEventVDM`** is passed on to the callback function when the event occurs. For details, see the description of **`OnEventVDM`**.

Note: This function sets the event generation mechanism to be a callback function, replacing any previously defined event notification mechanism. Only one event notification mechanism can be used at one time.

OnEvent

Windows9x and Windows2000 Only !	
Syntax	`INT WINAPI OnEvent(DevHandleT devHandle, HWND hWnd, OpaqueP lParam);`
	`devHandle` refers to either an interface or an external device.
	`hWnd` is the window handle to receive the event notification.
	`lParam` value will be passed in the notification message.
Returns	`-1` if error
Bus States	None

Example	`ieee = OpenName ("ieee");`
	`OnEvent (ieee, hWnd, (OpaqueP) 12345678L);`
	`Arm (ieee, acSRQ | acError);`
	`break;`

See Also	`OnEventVDM, Arm, Disarm`

The **`OnEvent`** command causes the event handling mechanism to issue a message upon occurrence of an **`Arm`**ed event.
The message will have a type of **`WM_IEEE488EVENT`**, whose value is retrieved via:

RegisterWindowMessage ((LPSTR) "WM_IEEE488EVENT");

The associated **`wParam`** is an event mask indicating which **`Arm`**ed event(s) caused the notification, and the **`lParam`** is the value passed to **`OnEvent`**. Note that although there is a macro for **`WM_IEEE488EVENT`** in the header file for each language, this macro resolves to a function call and therefore cannot be used as a case label. The preferred implementation is to include a default case in the message handling case statement and directly compare the message ID with **`WM_IEEE488EVENT`**.
The following is an example of an event handler.

```
LONG FAR WINAPI export
        WndProc(HWND hWnd, unsigned iMessage, WORD wParam, LONG lParam);
        {
        HANDLE
        ieee;
        switch (iMessage)
                {
                case WM_CREATE:
                    ieee = OpenName ("ieee");
                    OnEvent (ieee, hWnd, (OpaqueP) 12345678L);
                    Arm (ieee, acSRQ | acError);
                    break;
                default:
                    if (iMessage == WM_IEEE488EVENT) {
                       char buff [80];
                       wsprintf (buff, "Condition = %04X, Param = %081X",wParam, lParam);
                       MessageBox (hWnd, (LPSTR) buff, (LPSTR) "Event Noted", MB OK);
                       return TRUE;
                    }
                }
}
```

OnEventVDM

Windows9x and Windows2000 Only !	
Syntax	**`INT WINAPI OnEventVDM(DevHandleT devHandle, EventFuncT func);`**
	`devHandle` refers to either an interface or an external device.
	`func` is a user-specified interrupt-handler function that is to perform some user-defined function, when one of the **`Arm`**ed conditions occur.
Returns	**`-1`** if error
Bus States	None
Example	**`Arm(ieee0, acSRQ);`** **`OnEventVDM(ieee0, srqHandler);`** — Arms **`SRQ`** detection and sets up **`SRQ`** function handler
See Also	**`OnEvent, Arm, Disarm`**

This function is new in Driver488/W95. The **`OnEventVDM`** allows a call back to a user-specified function in an application. The following is a full example of a console mode program using the **`OnEventVDM`** function:

```
#include <windows.h>
#include <stdio.h>
#include “iotieee.h”
// For debugging
#define qsk(v,x) (v=x, printf(#x “returned %d\n”, v))
void srqHandler(DevHandleT devHandle, UINT mask)
{
        LONG xfered;
        printf(“\007\n\nEVENT-FUNCTION on %d mask 0x%04x\n”, devHandle, mask);
        qsk(xfered, Spoll(devHandle));
        printf(“\n\n”);
}
void main(void)
{
        LONG result, xfered;
        int ioStatus, x;
        DevHandleT ieee0, wave14, wave16;
        TermT myTerm;
        UCHAR buffer[500];
        printf(“\n\nSRQTEST program PID %d\n”,GetCurrentProcessId ());
        qsk(ieee0, OpenName(“ieee0”));
        qsk(wave14, OpenName(“Wave14”));
        qsk(wave16, OpenName(“Wave16”));
        qsk(result, Abort(wave14));
        qsk(result, Abort(wave16));
        qsk(x, Hello(ieee0, buffer));
        printf(“\n%s\n\n”, buffer);
        myTerm.EOI = 1;
        myTerm.nChar = 0;
        myTerm.termChar[0] = ‘\r’;
        myTerm.termChar[1] = ‘\n’;
        // Arm SRQ detection and set up SRQ function handler
        qsk(x, Arm(ieee0, acSRQ));
        qsk(x, OnEventVDM(ieee0, srqHandler));
        // Tell the Wave to assert SRQ in 3 seconds
        qsk(xfered,Output(wave16,“t3000x”,6L,1,0,&myTerm,0,&ioStatus));
        printf(“Completion code: 0x%04x\n”, ioStatus);
        // Normally, your program would be off doing other work, for
        // this example we will just hold here for a short time.
        For(result = 0; result 30000; result++) {
                printf(“Result is %06d\r”, result);
                }
            printf("\n\n");
            qsk(xfered, Spoll(wave16));
            qsk(x, Close(wave14));
            qsk(x, Close(wave16));
            qsk(x, Close(ieee0));
    }
```

******* Do we add a note for NT users on how to accomplish this?**

(Spoll interface for bit 7 in thread)

If so, what should the note say?

OpenName

Syntax	**`DevHandleT WINAPI OpenName(LPSTR name);`**
	`name` is the name of an interface or external device.
Returns	**`-1`** if error
	otherwise, the device handle associated with the given name
Bus State	None
Examples	**`dmm = OpenName("DMM");`** Opens the external device **`DMM`**
	`dmm = OpenName("IEEE:DMM");` Specifies the interface to which the external device is connected
See Also	**`MakeDevice, Close`**

The **`OpenName`** command opens the specified interface or external device and returns a device handle for use in accessing that device.

OutputX

Syntax	`LONG WINAPI OutputX(DevHandleT devHandle, LPBYTE data, DWORD count, BOOL last, BOOL forceAddr, TermT *term, BOOL reserved, LPDWORD compStat);`
	`devHandle` refers to either an interface or an external device. If `devHandle` refers to an external device, the `OutputX` command acts on the hardware interface to which the external device is attached.
	`data` is a string of bytes to send.
	`count` is the number of bytes to send.
	`last` is a flag that forces the device output terminator to be sent with the data.
	`forceAddr` is used to specify whether the addressing control bytes are to be issued for each `OutputX` command.
	`term` is a pointer to a terminator structure that is used to set up the input terminators. If `term` is set to `0`, the default terminator is used.
	`reserved` - this value is ignored.
	`compStat` is a pointer to an integer containing completion status information.
Returns	`-1` if error
	otherwise, the number of characters transferred
Bus States	With interface handle: `REN (if SC and AutoRemote), *ATN, ATN`
	With external device handle: `REN (if SC and AutoRemote), ATN•MTA, UNL, LAG, *ATN, ATN`

Example	`term.EOI = TRUE;`
	`term.nChar = 1;`
	`term.EightBits = TRUE;`
	`term.termChar[0] = '\r';`
	`data = "U0X";`
	`count = strlen(data);`
	`bytecnt=Output(timer,data,count,1,0,&term,0,&stat);`

See Also	`EnterX, Term, TimeOut, Buffered`

The **`OutputX`** command sends data to an interface or external device. The Remote Enable (**`REN`**) line is first asserted if Driver488 is the System Controller and **`AutoRemote`** is enabled. Then, if a device address (with optional secondary address) is specified, Driver488 is addressed to Talk and the specified device is addressed to Listen. If no address is specified, then Driver488 must already be configured to send data, either as a result of a preceding **`OutputX`** command, or as the result of a **`Send`** command. Terminators are automatically appended to the output data as specified.

The **`forceAddr`** flag is used to specify whether the addressing control bytes are to be issued for each **`OutputX`** command. If the device handle refers to an interface, **`forceAddr`** has no effect and command bytes are not sent. If the device handle refers to an external device and **`forceAddr`** is **`TRUE`**, Driver488 addresses the interface to Talk and the external device to Listen. If **`forceAddr`** is **`FALSE`**, Driver488 compares the current device with the most recently addressed device on that interface. If the addressing information is the same, no command bytes are sent. If they are different, **`OutputX`** acts as if the **`forceAddr`** flag were **`TRUE`** and sends the addressing information.

`term` is a pointer to a terminator structure that is used to set up the input terminators. This pointer may be a null pointer, requesting use of the default terminators for the device, or it may point to a terminator structure requesting no terminators.

The **`compStat`** is a pointer to an integer containing completion status information. A null pointer indicates that completion status is not requested.

Additional Output Functions
Driver488 provides additional `Output` functions that are short-form versions of the `OutputX` function. The following `Output` functions are already defined in your header file.

Output

Syntax	`LONG WINAPI Output(DevHandleT devHandle,LPBYTE data);`
Remarks	`Output` is equivalent to the following call to `OutputX`: `OutputX(devHandle,data,strlen(data),1,1,0L,0,0L);`

The `Output` function passes the device handle and data buffer to the `OutputX` function. It determines the size of the data buffer that you provided, and passes that value as the `count` parameter. It specifies that the `forceAddr` flag is set `TRUE`, which causes Driver488 to address the device if an external device is specified. The default terminators are chosen by specifying a `0` pointer as the `terminator` parameter. Synchronous transmission is specified by sending `0` for the `async` parameter, and the completion status value is ignored by sending a `0` for the `compStat` pointer.

OutputN

Syntax	`LONG WINAPI OutputN(DevHandleT devHandle,LPBYTE data,DWORD count);`
Remarks	`OutputN` is equivalent to the following call to `OutputX`: `OutputX(devHandle,data,count,0,1,0L,0,0L);`

The `OutputN` function passes the device handle and a pointer to the data buffer to the `OutputX` function. It specifies that the `forceAddr` flag is set `TRUE`, which causes Driver488 to address the device if an external device is specified. The default terminators are chosen by specifying a `0` pointer as the `terminator` parameter. Synchronous transmission is specified by sending `0` for the `async` parameter, and the completion status value is ignored by sending a `0` for the `compStat` pointer.

OutputMore

Syntax	`LONG WINAPI OutputMore(DevHandleT devHandle, LPBYTE data);`
Remarks	`OutputMore` is equivalent to the following call to `OutputX`: `OutputX(devHandle,data,strlen(data),1,0,0L,0,0L);`

The `OutputMore` function passes the device handle and data buffer to the `OutputX` function. It determines the size of the data buffer that you provided, and passes that value as the `count` parameter. It specifies that the `forceAddr` flag is set `FALSE`, so Driver488 does not re-address the device if it is the same device as that previously used. The default terminators are chosen by specifying a `0` pointer as the `terminator` parameter. Synchronous transmission is specified by sending `0` for the `async` parameter, and the completion status value is ignored by sending a `0` pointer for the `compStat` pointer.

OutputNMore

Syntax	`LONG WINAPI OutputNMore (DevHandleT devHandle, LPBYTE data, DWORD count);`
Remarks	`OutputNMore` is equivalent to the following call to `OutputX`: `OutputX(devHandle,data,0,0,0L,0,0L);`

The `OutputNMore` function passes the device handle and a pointer to the data buffer to the `OutputX` function. It specifies that the `forceAddr` flag is set `FALSE`, so Driver488 does not re-address the device if it is the same device as that previously used. The default terminators are chosen by specifying a `0` pointer as the `terminator` parameter. Synchronous transmission is specified by sending `0` for the `async` parameter, and the completion status value is ignored by sending a `0` pointer for the `compStat` pointer.

PPoll

Syntax	**`INT WINAPI PPoll(DevHandleT devHandle);`**
	`devHandle` refers to either an interface or an external device. If **`devHandle`** refers to an external device, then the **`PPoll`** command acts on the hardware interface to which the external device is attached.
Returns	**`-1`** if error
	otherwise, a number in the range 0 to 255
Bus States	**`ATN•EOI, *EOI`**
Example	**`errorcode = PPoll(ieee);`**
See Also	**`PPollConfig, PPollUnconfig, PPollDisable, SPoll`**

The **`PPoll`** (Parallel Poll) command is used to request status information from many bus devices simultaneously. If a device requires service then it responds to a Parallel Poll by asserting one of the eight IEEE 488 bus data lines (**`DIO1`** through **`DIO8`**, with **`DIO1`** being the least significant). In this manner, up to eight devices may simultaneously be polled by the controller. More than one device can share any particular **`DIO`** line. In this case, it is necessary to perform further Serial Polling (**`SPoll`**) to determine which device actually requires service.

Parallel Polling is often used upon detection of a Service Request (**`SRQ`**), though it may also be performed periodically by the controller. In either case, **`PPoll`** responds with a number from **`0`** to **`255`** corresponding to the eight binary **`DIO`** lines. Not every device supports Parallel Polling. Refer to the manufacturer's documentation for each bus device to determine if **`PPoll`** capabilities are supported.

PPollConfig

Syntax	**`INT WINAPI PPollConfig(DevHandleT devHandle,BYTE ppresponse);`**
	`devHandle` refers to either an interface or an external device to configure for the Parallel Poll.
	`ppresponse` is the decimal equivalent of the four binary bits **`S`**, **`P2`**, **`P1`**, and **`P0`** where **`S`** is the Sense bit, and **`P2`**, **`P1`**, and **`P0`** assign the **`DIO`** bus data line used for the response.
Returns	**-1** if error
Bus States	**`ATN•UNL, MTA, LAG, PPC`**
Example	**`errorcode = PPollConfig (dmm,0x0D);`** Configure device **`DMM`** to assert **`DIO6`** when it desires service (**`ist`** = 1) and it is Parallel Polled (0x0D = &H0D = 1101 binary; **`S`**=1, **`P2`**=1, **`P1`**=0, **`P0`**=1; 101 binary = 5 decimal = **`DIO6`**).
See Also	**`PPoll, PPollUnconfig, PPollDisable`**

The **`PPollConfig`** command configures the Parallel Poll response of a specified bus device. Not all devices support Parallel Polling and, among those that do, not all support the software control of their Parallel Poll response. Some devices are configured by internal switches.

The Parallel Poll response is set by a four-bit binary number response: **`S`**, **`P2`**, **`P1`**, and **`P0`**. The most significant bit of response is the *Sense* (**`S`**) bit. The Sense bit is used to determine when the device will assert its Parallel Poll response. Each bus device has an internal individual status (**`ist`**). The Parallel Poll response is asserted when this **`ist`** equals the Sense bit value **`S`**. The **`ist`** is normally a logic **`1`** when the device requires attention, so the **`S`** bit should normally also be a logic **`1`**. If the **`S`** bit is **`0`**, then the device asserts its Parallel Poll response when its **`ist`** is a logic **`0`**. That is, it does not require attention. However, the meaning of **`ist`** can vary between devices, so refer to your IEEE 488 bus device documentation. The remaining 3 bits of response: **`P2`**, **`P1`**, and **`P0`**, specify which **`DIO`** bus data line is asserted by the device in response to a Parallel Poll. These bits form a binary number with a decimal value from **`0`** through **`7`**, specifying data lines **`DIO1`** through **`DIO8`**, respectively.

PPollDisable

Syntax	**`INT WINAPI PPollDisable(DevHandleT devHandle);`**
	`devHandle` is either an interface or an external device that is to have its Parallel Poll response disabled.
Returns	**`-1`** if error
Bus States	**`ATN•UNL, MTA, LAG, PPC, PPD`**
Example	**`errorcode = PPollDisable(dmm);`** Disables Parallel Poll of device **`DMM`**.
See Also	**`PPoll, PPollConfig, PPollUnconfig`**

The **`PPollDisable`** command disables the Parallel Poll response of a selected bus device.

PPollDisableList

Syntax	`INT WINAPI PPollDisableList(DevHandlePT devHandleList);`
	`devHandleList` is a pointer to a list of external devices that are to have their Parallel Poll response disabled.
Returns	`-1` if error
Bus States	`ATN•UNL, MTA, LAG, PPC, PPD`

Example	`deviceList[0] = wave;`
	`deviceList[1] = timer;`
	`deviceList[2] = dmm;`
	`deviceList[3] = NODEVICE;`
	`errorcode = PPollDisableList(deviceList);`

See Also	`PPoll, PPollConfig, PPollUnconfig`

The **`PPollDisableList`** command disables the Parallel Poll response of selected bus devices.

PPollUnconfig

Syntax	**`INT WINAPI PPollUnconfig(DevHandleT devHandle);`**
	`devHandle` refers to a hardware interface. If **`devHandle`** refers to an external device, then the **`PPollUnconfig`** command acts on the hardware interface to which the external device is attached.
Returns	**`-1`** if error
Bus States	**`ATN•PPU`**
Example	**`errorcode = PPollUnconfig(ieee);`**
See Also	**`PPoll, PPollConfig, PPollDisable`**

The **`PPollUnconfig`** command disables the Parallel Poll response of all bus devices.

Remote

Syntax	**`INT WINAPI Remote(DevHandleT devHandle);`**	
	`devHandle` refers to either an interface or an external device. If **`devHandle`** refers to an interface, then the Remote Enable (**`REN`**) line is asserted. If **`devHandle`** refers to an external device, then that device is addressed to Listen and placed into the **`Remote`** state.	
Returns	**`-1`** if error	
Bus States	With interface: **`REN`**	
	With external device: **`REN, ATN•UNL, MTA, LAG`**	
Examples	**`errorcode = Remote(ieee);`**	Asserts the **`REN`** bus line
	`errorcode = Remote(scope);`	Asserts the **`REN`** bus line and addresses the **`scope`** device specified to Listen, to place it in the **`Remote`** state
See Also	**`Local, LocalList, RemoteList`**	

The **`Remote`** command asserts the Remote Enable (**`REN`**) bus management line. If an external device is specified, then **`Remote`** will also address that device to Listen, placing it in the **`Remote`** state.

RemoteList

Syntax	`INT WINAPI RemoteList(DevHandlePT devHandleList);`	
	`devHandleList` is a pointer to a list of devices.	
Returns	`-1` if error	
Bus States	`REN, ATN•UNL, MTA, LAG`	

Example	`deviceList[0] = wave;`
	`deviceList[1] = timer;`
	`deviceList[2] = dmm;`
	`deviceList[3] = NODEVICE;`
	`errorcode = RemoteList(deviceList);`
		Asserts the **REN** bus line and addresses a list of specified devices to Listen, to place these specified devices in the **Remote** state.

See Also	`Remote, Local, LocalList`	

The **RemoteList** command asserts the Remote Enable (**REN**) bus management line. If external devices are specified, then **RemoteList** will also address those devices to Listen, placing them in the **Remote** state.

RemoveDevice

Syntax	**`INT WINAPI RemoveDevice(DevHandleT devHandle);`**
	`devHandle` specifies an interface or an external device to remove.
Returns	**`-1`** if error
Bus States	None
Example	**`errorcode = RemoveDevice(dmm);`**
See Also	**`MakeDevice, KeepDevice`**

The **`RemoveDevice`** command removes the specific temporary or permanent Driver488 device that was created with either the **`MakeDevice`** command or the startup configuration. This command also removes a device that was made permanent through a **`KeepDevice`** command.

Reset

Syntax	`INT WINAPI Reset(DevHandleT devHandle);`
	`devHandle` refers to either an interface or an external device. If `devHandle` refers to an external device, the `Reset` command acts on the hardware interface to which the external device is attached.
Returns	`-1` if error
Bus States	None
Example	`errorcode=Reset(ieee);`
See Also	`Abort, Term, TimeOut`

The `Reset` command provides a warm start of the interface. Using **Reset** is equivalent to issuing the following command process, including clearing all error conditions.

Disarm
Reset hardware (resets to Peripheral if not System Controller)
Abort (if System Controller)
Error ON
Local
Request 0 (if Peripheral)
Clear **Change**, **Trigger**, and **Clear** status
Reset I/O adapter settings to installed values (**BusAddress**, **TimeOut**, **IntLevel** and **DmaChannel**)

SendCmd

Syntax	`INT WINAPI SendCmd(DevHandleT devHandle, LPBYTE commands, DWORD count);`
	`DevHandle` refers to an interface handle.
	`Commands` points to a string of command bytes to be sent.
	`count` is the length of the command string.
Returns	**-1** if error
Bus States	User-defined

Example	`char command[] = “U?0";`
	`errorcode = SendCmd(ieee, &command, sizeof command);`

See Also	`SendData, SendEoi`

The **`SendCmd`** command sends a specified string of bytes with Attention (**`ATN`**) asserted, causing the data to be interpreted as IEEE 488 command bytes.

SendData

Syntax	`INT WINAPI SendData(DevHandleT devHandle, LPBYTE data, DWORD count);`
	`DevHandle` refers to an interface handle.
	`data` points to a string of data bytes to be sent.
	`count` is the length of the data string.
Returns	**-1** if error
Bus States	User-defined

Example	`char data[] = “W0X”;`
	`errorcode = SendData(ieee, data, strlen (data));`

See Also	`SendCmd, SendEoi`

The **`SendData`** command provides byte-by-byte control of data transfers and gives greater flexibility than the other commands. This command can specify exactly which operations Driver488 executes.

SendEoi

Syntax	`INT WINAPI SendEoi(DevHandleT devHandle, LPBYTE data, DWORD count);`
	`DevHandle` refers to an interface handle.
	`data` points to a string of data bytes to be sent.
	`Count` is the length of the data string.
Returns	**-1** if error
Mode	Any
Bus States	User-defined
Example	`char data[] = “W0X”;` `errorcode = SendEoi(ieee, data, strlen (data));`
See Also	`SendCmd, SendData`

The `SendEoi` command provides byte-by-byte control of data transfers and gives greater flexibility than the other commands. This command can specify exactly which operations Driver488 executes. Driver488 asserts `EOI` during the transfer of the final byte.

SPoll

Syntax	`INT WINAPI SPoll(DevHandleT devHandle);`	
	`DevHandle` refers to either an interface or a specific external device.	
Returns	`-1` if error	
	otherwise, `0` or `64` (hardware interface) in the range `0` to `255` (external device)	
Bus States	`ATN•UNL, MLA, UNT, TAG, SPE, *ATN, ATN•SPD, UNT`	
Examples	`errorcode = SPoll(ieee);`	Returns the internal `SRQ` status
	`errorcode = SPoll(dmm);`	Returns the Serial Poll response of the specified device
See Also	`SPollList, PPoll`	

In Active Controller mode, the **`SPoll`** (Serial Poll) command performs a Serial Poll of the bus device specified and responds with a number from **`0`** to **`255`** representing the decimal equivalent of the eight-bit device response. If **`rsv`** (**`DIO7`**, decimal value **`64`**) is set, then that device is signaling that it requires service. The meanings of the other bits are device-specific.

Serial Polls are normally performed in response to assertion of the Service Request (**`SRQ`**) bus signal by some bus device. In Active Controller mode, with the interface device specified, the **`SPoll`** command returns the internal **`SRQ`** status. If the internal **`SRQ`** status is set, it usually indicates that the **`SRQ`** line is asserted. Driver488 then returns a **`64`**. If it is not set, indicating that **`SRQ`** is not asserted, then Driver488 returns a **`0`**. With an external device specified, **`SPoll`** returns the Serial Poll status of the specified external device.

In Peripheral mode, the **`SPoll`** command is issued only to the interface, and returns the Serial Poll status. If **`rsv`** (**`DIO7`**, decimal value **`64`**) is set, then Driver488 has not been Serial Polled since the issuing last **`Request`** command. The **`rsv`** is reset whenever Driver488 is Serial Polled by the Controller.

SPollList

Syntax	`INT WINAPI SPollList(DevHandlePT devHandleList, LPBYTE resultList, BYTE SPollFlag);`
	`devHandleList` is a pointer to a list of external devices.
	`resultList` is an array that is filled in with the Serial Poll results of the corresponding external devices.
	`SPollFlag` refers to either `ALL`, `WHILE_SRQ`, or `UNTIL_RSV`.
Returns	`-1` if error
Bus States	`ATN•UNL, MLA, TAG, SPE, *ATN, ATN•SPD, UNT`
Example	`deviceList[0] = wave;` `deviceList[1] = timer;` `deviceList[2] = dmm;` `deviceList[3] = NODEVICE;` `result = SPollList(deviceList, resultList, ALL);` — Returns the Serial Poll response for a list of device handles.
See Also	`SPoll, PPoll`

In Active Controller mode, the **`SPollList`** (Serial Poll) command performs a Serial Poll of the bus devices specified and responds with a number from `0` to `255` (representing the decimal equivalent of the eight-bit device response) for each device on the list. If **`rsv`** (`DIO7`, decimal value `64`) is set, then that device is signaling that it requires service. The meanings of the other bits are device-specific.

Serial Polls are normally performed in response to assertion of the Service Request (**`SRQ`**) bus signal by some bus device. In Active Controller mode with the interface device specified, the **`SPollList`** command returns the internal **`SRQ`** status for each device. If the internal **`SRQ`** status is set, it usually indicates that the **`SRQ`** line is asserted. Driver488 then returns a `64`. If it is not set, indicating that **`SRQ`** is not asserted, then Driver488 returns a `0`. With an external device specified, **`SPollList`** returns the Serial Poll status of the specified external device.

In Peripheral mode, the **`SPollList`** command is issued only to the interface and returns the Serial Poll status. If **`rsv`** (`DIO7`, decimal value `64`) is set, then Driver488 has not been Serial Polled since the last **`Request`** command was issued. The **`rsv`** is reset whenever Driver488 is Serial Polled by the Controller.

The **`SPollFlag`** refers to either **`ALL`**, **`WHILE_SRQ`**, or **`UNTIL_RSV`**. If **`ALL`** is chosen, all the devices are Serial Polled and their results placed into the result array. If **`SPollFlag`** is **`WHILE_SRQ`**, Driver488 Serial Polls the devices until the **`SRQ`** bus signal becomes unasserted, and the results are put into the result array. If **`SPollFlag`** is **`UNTIL_RSV`**, Driver488 Serial Polls the devices until the first device whose **`rsv`** bit is set, is found and the results are put into the result array.

Status

Syntax	**`INT WINAPI Status(DevHandleT devHandle, IeeeStatusPT result);`**
	`devHandle` refers to either an IEEE 488 interface or an external device. If **`devHandle`** refers to an external device, **`Status`** acts on the hardware interface to which the external device is attached.
	`result` is a pointer to a **`Status`** structure.
Returns	`-1` if error
Bus States	None

Example	`result = Status(ieee,&StatusResult);`
	`if (StatusResult.SRQ) {`
	`printf("We have a serial poll request");`
	`} else {`
	`printf("There is no serial poll request");`
	`}`

See Also	**`GetError, SPoll`**

The **`Status`** command returns various items detailing the current state of Driver488. They are returned in a data structure, based on the following table.

Status Item	**Structure Menber**	**Values and Description**
Primary Bus Address	.Primaddr	`0` to `30`: Two-digit decimal number.
Service Request	.SRQ	**`TRUE`**: SRQ is asserted, **`FALSE`**: SRQ is not asserted.

The *Primary Bus Address* (**`.Primaddr`**) is the IEEE 488 bus device primary address assigned to Driver488 or the specified device. This will be an integer from `0` to `30`.

The *Service Request* field (**`.SRQ`**), as an active controller, reflects the IEEE 488 bus **`SRQ`** line signal. As a peripheral, this status reflects the **`rsv`** bit that can be set by the **`Request`** command and is cleared when the Driver488 is Serial Polled. For more details, refer to the **`SPoll`** command in this chapter.

Talk

Syntax	`INT WINAPI Talk(DevHandleT devHandle, BYTE primary, BYTE secondary);`
	`devHandle` refers to either an interface or an external device. If **`devHandle`** refers to an external device, the **`Talk`** command acts on the associated interface.
	`primary` and **`secondary`** specify the primary and secondary addresses of the device which is to be addressed to Talk.
Returns	**`-1`** if error
Bus States	**`ATN, TAG`**
Example	**`errorcode = Talk (ieee, 12, -1);`**
See Also	**`Listen, SendCmd`**

The **`Talk`** command addresses an external device to Talk.

Term

Syntax	`INT WINAPI Term(DevHandleT devHandle, TermT *term, DWORD termFlag);`
	`devHandle` refers to either an interface or an external device.
	`term` is a pointer to the terminator structure.
	`termFlag` can be either **`TERMIN`**, **`TERMOUT`**, or **`TERMIN+TERMOUT`**, specifying whether input, output, or both are being set.
Returns	`-1` if error
Bus States	None

Example	`term.EOI = TRUE;`
	`term.nChar = 1;`
	`term.EightBits = TRUE;`
	`term.termChar[0] = 13;`
	`errorcode = Term(ieee,&term,TERMIN);`

See Also	`TermQuery, EnterX, OutputX, Status`

The **`Term`** command sets the end-of-line (**`EOL`**) terminators for input from, and output to, I/O adapter devices. These terminators are sent at the end of output data and expected at the end of input data, in the manner of **`CR LF`** as used with printer data.

During output, **`Term`** appends the bus output terminator to the data before sending it to the I/O adapter device. Conversely, when Driver488 receives the bus input terminator, it recognizes the end of a transfer and returns the data to the calling application. The terminators never appear in the data transferred to or from the calling application. The default terminators for both input and output are set by the startup configuration and are normally **`CR LF EOI`**, which is appropriate for most bus devices.

End-Or-Identify (**`EOI`**) has a different meaning when it is specified for input than when it is specified for output. During input, **`EOI`** specifies that input is terminated on detection of the **`EOI`** bus signal, regardless of which characters have been received. During output, **`EOI`** specifies that the **`EOI`** bus signal is to be asserted during the last byte transferred.

TermQuery

Syntax	`INT TermQuery(DevHandleT devHandle, TermT *term, INT termFlag);`
	`devHandle` refers to either an interface or an external device.
	`terminator` is a pointer to the terminator structure.
	`termFlag` can be either **`TERMIN`**, **`TERMOUT`**, or **`TERMIN+TERMOUT`**, specifying whether input, output, or both are being set.
Returns	`-1` if error
Bus States	None
Example	None provided.
See Also	**`Term, EnterX, OutputX, Status`**

This is a new function in Driver488/W95. The **`TermQuery`** function queries the terminators setting. Terminators are defined by the **`TermT`** structure.

TimeOut

Syntax	**`INT WINAPI TimeOut(DevHandleT devHandle, DWORD millisec);`**
	`devHandle` refers to either an IEEE 488 interface or an external device.
	`millisec` is a numeric value given in milliseconds.
Returns	**`-1`** if error
Bus States	None
Example	**`errorcode = TimeOut(ieee,100);`** Sets the timeout value to 100 msec.
See Also	**`TimeOutQuery, Reset`**

The **`TimeOut`** command sets the number of milliseconds that Driver488 waits for a transfer before declaring a timeout error. The timeout value sets a limit on the total time allowed for an operation. For example, when using the **Enter** command all expected data must be received within the specified timeout period or a timeout error will occur.

Timeout checking may be suppressed by specifying a timeout value of zero seconds. The zero seconds specification results in a timeout period of about 49 days.

TimeOutQuery

Syntax	`INT WINAPI TimeOutQuery(DevHandleT devHandle, DWORD *millisec);`
	`devHandle` refers to either an IEEE 488 interface or an external device.
	`millisec` is a pointer to a buffer that is to receive the timeout value, given in milliseconds.
Returns	`-1` if error
Bus States	None
Example	None provided.
See Also	`TimeOut, Reset`

This is a new function in Driver488/W95. The `TimeOutQuery` function queries the timeout setting, given in milliseconds.

Trigger

Syntax	**`INT WINAPI Trigger(DevHandleT devHandle);`**
	`devHandle` refers to either an IEEE 488 interface or an external device.
Returns	**`-1`** if error
Bus States	With interface handle: **`ATN•GET`**
	With external device handle: **`ATN•UNL, MTA, LAG, GET`**
Examples	**`errorcode = Trigger(ieee);`** Issues a Group Execute Trigger (**`GET`**) bus command to those devices that are already in the Listen state as the result of a previous **`Output`** or **`Send`** command
	`errorcode = Trigger(dmm);` Issues a Group Execute Trigger (**`GET`**) bus command to the device specified
See Also	**`TriggerList, Status, SendCmd`**

The **`Trigger`** command issues a Group Execute Trigger (**`GET`**) bus command to the specified device. If no interface devices are specified, then the **`GET`** only affects those devices that are already in the Listen state as a result of a previous **`Output`** or **`Send`** command.

TriggerList

Syntax	**`INT WINAPI TriggerList(DevHandlePT devHandleList);`**
	`devHandleList` is a pointer to a list of external devices.
Returns	**`-1`** if error
Bus States	**`ATN•UNL, MTA, LAG, GET`**

Example	**`deviceList[0] = wave;`**
	`deviceList[1] = timer;`
	`deviceList[2] = dmm;`
	`deviceList[3] = NODEVICE;`
	`errorcode = TriggerList(deviceList);`
	Issues a Group Execute Trigger (**`GET`**) bus command to a list of specified devices.

See Also	**`Trigger, SendCmd, Status`**

The **`TriggerList`** command issues a Group Execute Trigger (**`GET`**) bus command to the specified devices. If no interface devices are specified, then the **`GET`** affects those devices that are already in the Listen state as a result of a previous **`Output`** or **`Send`** command.

UnListen

Syntax	**`INT WINAPI UnListen (DevHandleT devHandle);`**
	`devHandle` refers to either an interface or an external device. If **`devHandle`** refers to an external device, the **`UnListen`** command acts on the associated interface.
Returns	**`-1`** if error
Bus States	**`ATN, UNL`**
Example	**`errorcode = UnListen (ieee);`**
See Also	**`Listen, UnTalk, SendCmd, Status`**

The **`UnListen`** command unaddresses an external device that was addressed to Listen.

UnTalk

Syntax	**`INT WINAPI UnTalk (DevHandleT devHandle);`**
	`devHandle` refers to either an interface or an external device. If **`devHandle`** refers to an external device, the **`UnTalk`** command acts on the associated interface.
Returns	**`-1`** if error
Bus States	**`ATN, UNT`**
Example	**`errorcode = UnTalk (ieee);`**
See Also	**`Talk, UnListen, SendCmd, Status`**

The **`UnTalk`** command unaddresses an external device that was addressed to Talk.

Notes

Appendix A

API Error Codes

Error Number and Message Text		Description
00	OK	No error has occurred.
01	TIME OUT - NOT ADDRESSED TO LISTEN	ENTER as a Peripheral did not receive data within the TIME OUT period.
02	AUTOINITIALIZE MODE NOT ALLOWED	This error message is obsolete in Driver488 Rev.3.0.
03	SYSTEM ERROR - BUFFER MODE NOT SUPPORTED	Internal system error. Report to factory.
04	TIME OUT ERROR ON DATA READ	Expected bus data was not received within the TIME OUT period.
05	SYSTEM ERROR - INVALID INTERNAL MODE	Internal system error. Report to factory.
06	INVALID CHANNEL FOR DMA	This error message is obsolete in Driver488 Rev.4.0.
07	TIME OUT ON DMA TRANSFER	This error message is obsolete in Driver488 Rev.4.0.
08	TIME OUT - NOT ADDRESSED TO TALK	OUTPUT as a Peripheral was not possible within the TIME OUT period.
09	TIME OUT OR BUS ERROR ON WRITE	Error occurred transferring a data byte to a bus service.
10	SEQUENCE - NO DATA AVAILABLE	The user's program attempted to read from Driver488 when no response or data was available.
11	SEQUENCE - DATA HAS NOT BEEN READ	The user's program attempted to write data or commands to Driver488 without reading back the response to a previous command.
12	SYSTEM ERROR - ON PEN INTS ALREADY ON	Internal system error. Report to factory.
13	SYSTEM ERROR - INVALID ON PEN INIT	Internal system error. Report to factory.
14	SYSTEM ERROR - LIKELY MEMORY CORRUPTION	Internal system error. Report to factory.
15	SYSTEM ERROR - ON PEN INTS ALREADY OFF	Internal system error. Report to factory.
16	BOARD DOES NOT RESPOND AT SPECIFIED ADDRESS	Driver488 is unable to communicate with the IEEE interface board. Check the board address configuration, and the software installation.
17	TIME OUT ON COMMAND (MTA)	MyTalkAddress could not be sent within the TIME OUT period.
18	TIME OUT ON COMMAND (MLA)	MyListenAddress could not be sent within the TIME OUT period.
19	TIME OUT ON COMMAND (LAG)	Listen address(es) could not be sent within the TIME OUT period.
20	TIME OUT ON COMMAND (TAG)	Talk address could not be sent within the TIME OUT period.
21	TIME OUT ON COMMAND (UNL)	UnListen could not be sent within the TIME OUT period.
22	TIME OUT ON COMMAND (UNT)	UnTalk could not be sent within the TIME OUT period.
23	ONLY AVAILABLE TO SYSTEM CONTROLLER	Driver488 could not execute a command because it was not the System Controller.
24	RESPONSE MUST BE 0 THROUGH 15	The RESPONSE parameter of the PPOLL CONFIG command must be within the range of 0 to 15.
25	NOT A PERIPHERAL	The REQUEST command is only valid when Driver488 is in the Peripheral (*CA) mode.
26	SYSTEM ERROR - TIMER INTS ALREADY ON	Internal system error. Report to factory.
27	SYSTEM ERROR - INVALID TIMER INIT	Internal system error. Report to factory.
28	SYSTEM ERROR - TIMER INTS ALREADY OFF	Internal system error. Report to factory.
29	ADDRESS REQUIRED	PASS CONTROL requires an address.
30	TIME OUT VALUE MUST BE FROM 0 TO 65535	The TIME OUT period must be within the specified range.

31	MUST BE ADDRESSED TO TALK	DATA or EOI SEND subcommands are invalid unless Driver488 is already addressed to talk by MTA.
32	VALUE MUST BE BETWEEN 0 AND 255	Data bytes specified numerically in the SEND command must be 8-bit integers.
33	INVALID BASE ADDRESS	I/O port base addresses must end in 0, 1, or 8 when expressed in hexadecimal.
34	INVALID BUS ADDRESS	IEEE 488 bus addresses must be in the range of 0 to 30.
35	BAD DMA CHAN NO. OR DMA NOT ENABLED	This error message is obsolete in Driver488 Rev.4.0.
36	NOT AVAILABLE TO A PERIPHERAL	In Peripheral mode Driver488 cannot send bus commands such as device addresses.
37	INVALID PRIMARY ADDRESS	IEEE 488 bus addresses must be in the range of 0 to 30.
38	INVALID SECONDARY ADDRESS	IEEE 488 bus secondary addresses must be in the range of 0 to 31.
39	INVALID - TRANSFER OF ZERO BYTES	A #count of zero bytes is not valid.
40	NOT ADDRESSED TO LISTEN	In Controller mode, ENTER without specifying a bus address is not valid unless Driver488 is already addressed to listen.
41	COMMAND SYNTAX ERROR	Error in specifying command.
42	UNABLE TO CHANGE MODE AFTER BOOTUP	This error message is obsolete in Driver488 Rev.4.0.
43	TIME OUT WAITING FOR ATTENTION	As a Peripheral, executing an ENTER command, Attention did not become unasserted within the TIME OUT period.
44	DEMO VERSION - CAPABILITY EXHAUSTED	The DEMO version of Driver488 is limited to 100 commands per session.
45	DEMO VERSION - ONLY ONE ADDRESS	The DEMO version of Driver488 can control only one instrument at one IEEE 488 bus address.
46	OPTION NOT AVAILABLE	This error message is obsolete in Driver488 Rev.4.0.
47	VALUE MUST BE BETWEEN 1 AND 8	The IEEE 488 interface board clock frequency must be between 1 and 8.
48	TIME OUT - CONTROL NOT ACCEPTED	No device took control of the IEEE 488 bus after a PASS CONTROL.
49	UNABLE TO ADDRESS SELF TO TALK OR LISTEN	A TALK or LISTEN subcommand in a SEND command specified the controller's own address. Use MTA or MLA instead.
50	TIME OUT ON COMMAND	A time out error occurred during a SEND command.
51	CANNOT DMA ON ODD BOUNDARY	Internal system error. Report to factory.
52	INTERRUPT %d DOES NOT EXIST	Invalid interrupt chosen. Check hardware settings.
53	INTERRUPT %d IS NOT SHAREABLE	Another device already controls this interrupt.
54	UNABLE TO ALLOCATE DYNAMIC MEMORY FOR INT %d	Internal system error. Report to factory.
55	SHARED INTERRUPT %d CHAIN CORRUPTED	Internal system error. Report to factory.
56	TOO MANY ACTIVE TIMEOUTS	Internal system error. Report to factory.
57	INVALID DEVICE HANDLE %d	Device handle was not opened. Must first open device and assign handle.
58	OUT OF DEVICE HANDLES	Too many device handles opened. Must close unused handles.
59	UNKNOWN DEVICE: %s	Device not configured. Use MakeDevice to create.
60	DRIVER NOT LOADED	Driver is not loaded. Must load driver to run.
61	INVALID LIST OF DEVICE HANDLES	Array of device handles does not contain valid handles.
62	INVALID TERMINATOR STRUCTURE	Terminator structure does not contain valid data.
63	INVALID DATA POINTER	Data pointer is NULL or points to invalid data.
64	INVALID POINTER TO STATUS STRUCTURE	Status structure address is invalid or NULL.
65	INVALID NAME POINTER	Name parameter is empty or address is invalid.
66	SYSTEM ERROR - INVALID INTERNAL POINTER	Internal system error. Report to factory.
67	INVALID STRING FOR ERROR TEXT	Error text string address is invalid.
68	UNABLE TO FIND ERROR CODE REPORTER	Internal system error. Report to factory.
69	UNABLE TO TRANSLATE ERROR CODE	Internal system error. Report to factory.

70	DMA CHANNEL %d DOES NOT EXIST	Specified DMA channel does not exist. Check hardware settings.
71	DMA CHANNEL %d NOT AVAILABLE	Specified DMA channel is not available for use by Driver. Choose another channel.
72	DMA CHANNEL %d ALREADY IN USE	Specified DMA channel is already being used by another device. Choose another channel.
73	UNABLE TO ALLOCATE MEMORY FOR ASYNCHRONOUS I/O	Internal system error. Report to factory.
74	UNKNOWN DOS DEVICE NAME	Driver488 DOS device name not known. Must create Driver488 DOS device name with the Make Dos Name command.
75	UNABLE TO ALLOCATE MEMORY FOR NEW DEVICE	Ran out of memory. Remove some devices to restore memory.
76	UNKNOWN SLAVE DEVICE	Internal system error. Report to factory.
77	SLAVE DEVICE NOT SPECIFIED	Corrupt initialization file. Run the Install program.
78	UNABLE TO CREATE DOS DEVICE NAME	Internal system error. Report to factory.
79	UNABLE TO INITIALIZE DEVICE	Corrupt initialization file. Run the Install program.
80	ATTEMPTED TO REMOVE SLAVE DEVICE	Attempt to remove device which is required for operation by another device.
81	DATA OVERRUN	Serial input overflow.
82	(None)	(None)
83	FRAMING ERROR	Serial data corrupt. Possible incorrect Serial port configuration.
84	TIME OUT ON SERIAL COMMUNICATION	Serial device did not respond.
85	UNKNOWN PARAMETER OF TYPE %d SPECIFIED :\n %s = %s	Internal system error. Report to factory.
86	BUS ERROR - NO LISTENERS	No Listeners found on bus.
87	TIME OUT ON MONITOR DATA	Expected terminator was not received.
88	INVALID VALUE SPECIFIED	Specified value is invalid for application. See command for proper value types.
89	NO TERMINATOR SPECIFIED	Terminator must be specified. Check terminator value.
90	NOT AVAILABLE IN 8-BIT SLOT	Specified option is not available when the I/O adapter is fitted into an 8-bit slot.
91	TOO MANY PENDING EVENTS	Internal system error. Report to factory.
92	BREAK ERROR	Serial receiver detected break.
93	UNEXPECTED CHANGE OF CONTROL LINES	Handshake lines changed during transmission.
94	TIME OUT ON CTS	Hardware handshake not satisfied within time out. Check cabling and connected device.
95	TIME OUT ON DSR	Hardware handshake not satisfied within time out. Check cabling and connected device.
96	TIME OUT ON DCD	Hardware handshake not satisfied within time out. Check cabling and connected device.
97	CANNOT SEND EOI WITHOUT DATA	Transmission of EOI was requested when no data was available to send.
98	ADDRESS STATUS CHANGE DURING TRANSFER	Talker/Listener mode changed during data transfer, possibly due to activity of some other device on the IEEE 488 bus.
99	UNABLE TO MAKE NEW DEVICE	MakeDevice or CreateDevice was unable to create a new device, possibly due to the number of devices which already existed.
100	(None)	(None)
101	COMMAND SYNTAX ERROR: %	Command interpreter was unable to interpret command and no other information was available.
102	ERROR OPENING DEVICE %s	Possible loss of electrical or logical connection.
103	DEVICE %s CURRENTLY LOCKED BY %s	Device is in use by another processor in a multiprocessing environment. (This does not refer to multitasking environments.)

104	TIME OUT ON NETWORK COMMUNICATIONS	Unable to access a remote communications device within the time out interval.
105	ERROR: DEVICE IS NOT OPEN	Attempt to access a device which has not been opened or has subsequently been closed.
106	IPX IS NOT LOADED	Unable to access device via network communications.
107	INTERFACE IS BUSY	Remote IEEE 488 interface is busy.
108	TIMER/COUNTER REQUIRES INTERRUPTS TO BE CONFIGURED	Interrupts are required for proper Driver488 function of this device.
109	INVALID INTERRUPT LEVEL	Request interrupt level is not supported by this hardware.
110	MUST REMOVE DOS NAME FIRST	Attempted to remove a Driver488 device underlying a DOS device.
111	NO WINDOWS TIMERS AVAILABLE	Driver488/W31 requires use of a Windows timer that was unavailable. Close other applications using Windows timers and retry.
112	OBSOLETE LIBRARY FUNCTION	The called function is no longer supported.
113	UNABLE TO GET PROCEDURE ADDRESS	Thunking layer internal error.
114	MEMORY ALLOCATION ERROR	Thunking layer internal error.
115	FUNCTION ARGUMENT IS BAD READ POINTER	An invalid pointer argument was passed to a thunking layer function.
116	FUNCTION ARGUMENT IS BAD WRITE POINTER	An invalid pointer argument was passed to a thunking layer function.
117	OTHER THUNKING-LAYER ERROR	Thunking layer internal error.
118	THE NAMED DEVICE IS ALREADY OPEN	OpenName was called with the name of a device that is already open.
119	INVALID SYSTEM RESOURCE SETTING	Driver detected invalid PnP resource setting.
120	INTERFACE HARDWARE IS NOT PRESENT OR, HAS NOT BEEN SELECTED.	OpenName was called with the name of a device that has not been assigned to a hardware device.

Appendix B

ASCII Codes

B7 B6 B5 BITS B4 B3 B2 B1	Ø Ø Ø	Ø Ø 1	Ø 1 Ø	Ø 1 1	1 Ø Ø	1 Ø 1	1 1 Ø	1 1 1
	CONTROL		NUMBERS SYMBOLS		UPPER CASE		LOWER CASE	
Ø Ø Ø Ø	Ø NUL Ø Ø	2Ø DLE 1Ø 16	4Ø SP 2Ø 32	6Ø 0 3Ø 48	1ØØ @ 4Ø 64	12Ø P 5Ø 8Ø	14Ø ` 6Ø 96	16Ø p 7Ø 112
Ø Ø Ø 1	1 GTL SOH 1 1	21 LLO DC1 11 17	41 ! 21 33	61 1 31 49	1Ø1 A 41 65	121 Q 51 81	141 a 61 97	161 q 71 113
Ø Ø 1 Ø	2 STX 2 2	22 DC2 12 18	42 " 22 34	62 2 32 5Ø	1Ø2 B 42 66	122 R 52 82	142 b 62 98	162 r 72 114
Ø Ø 1 1	3 ETX 3 3	23 DC3 13 19	43 # 23 35	63 3 33 51	1Ø3 C 43 67	123 S 53 83	143 c 63 99	163 s 73 115
Ø 1 Ø Ø	4 SDC EOT 4 4	24 DCL DC4 14 2Ø	44 $ 24 36	64 4 34 52	1Ø4 D 44 68	124 T 54 84	144 d 64 1ØØ	164 t 74 116
Ø 1 Ø 1	5 PPC ENQ 5 5	25 PPU NAK 15 21	45 % 25 37	65 5 35 53	1Ø5 E 45 69	125 U 55 85	145 e 65 1Ø1	165 u 75 117
Ø 1 1 Ø	6 ACK 6 6	26 SYN 16 22	46 & 26 38	66 6 36 54	1Ø6 F 46 7Ø	126 V 56 86	146 f 66 1Ø2	166 v 76 118
Ø 1 1 1	7 BEL 7 7	27 ETB 17 23	47 ' 27 39	67 7 37 55	1Ø7 G 47 71	127 W 57 87	147 g 67 1Ø3	167 w 77 119
1 Ø Ø Ø	1Ø GET BS 8 8	3Ø SPE CAN 18 24	5Ø (28 4Ø	7Ø 8 38 56	11Ø H 48 72	13Ø X 58 88	15Ø h 68 1Ø4	17Ø x 78 12Ø
1 Ø Ø 1	11 TCT HT 9 9	31 SPD EM 19 25	51) 29 41	71 9 39 57	111 I 49 73	131 Y 59 89	151 i 69 1Ø5	171 y 79 121
1 Ø 1 Ø	12 LF A 1Ø	32 SUB 1A 26	52 * 2A 42	72 : 3A 58	112 J 4A 74	132 Z 5A 9Ø	152 j 6A 1Ø6	172 z 7A 122
1 Ø 1 1	13 VT B 11	33 ESC 1B 27	53 + 2B 43	73 ; 3B 59	113 K 4B 75	133 [5B 91	153 k 6B 1Ø7	173 { 7B 123
1 1 Ø Ø	14 FF C 12	34 FS 1C 28	54 , 2C 44	74 < 3C 6Ø	114 L 4C 76	134 \ 5C 92	154 l 6C 1Ø8	174 ¦ 7C 124
1 1 Ø 1	15 CR D 13	35 GS 1D 29	55 - 2D 45	75 = 3D 61	115 M 4D 77	135] 5D 93	155 m 6D 1Ø9	175 } 7D 125
1 1 1 Ø	16 SO E 14	36 RS 1E 3Ø	56 . 2E 46	76 > 3E 62	116 N 4E 78	136 ^ 5E 94	156 n 6E 11Ø	176 ~ 7E 126
1 1 1 1	17 SI F 15	37 US 1F 31	57 / 2F 47	77 ? 3F 63	117 O 4F Ø	137 _ 5F 95	157 o 6F 111	177 RUBOUT (DEL) 7F 127
	ADDRESSED COMMANDS	UNIVERSAL COMMANDS	LISTEN ADDRESSES		TALK ADDRESSES		SECONDARY ADDRESSES OR COMMANDS	

KEY:

octal	25 PPU	Message Mnemonic
	NAK	ASCII/ISO character
hex	15 21	decimal

ASCII 7-bit Code Chart

Notes

Appendix C

Troubleshooting IEEE 488 Systems and Software

To efficiently diagnose, troubleshoot and verify IEEE 488 systems, you should first have some basic knowledge of the IEEE bus. Since the hardware portion of the IEEE standard is rigorous and stable, most of the problems you will encounter during the system integration process will be in the application software. This note contains a brief IEEE tutorial followed by troubleshooting techniques.

The IEEE 488 Bus Standard

Addresses

Each device on the IEEE 488 bus has a unique address — even the controller. The addresses range from 0 to 30. Addresses are the means by which controllers select specific instruments. To send data to a device, the controller must both address itself to talk and address the device to listen.

Commands

IEEE 488 recognizes two types of commands: Device-Dependent Commands (DDCs) and IEEE 488-specific commands. DDC's such as “R0F0T2X” are simply data sent from one device to another (in this instance from the controller to an instrument). IEEE 488-specific commands come in two forms; multiline commands and uniline commands. Multiline commands are sent on the data bus lines; uniline commands are individual signals on the bus.

The uniline signals are the easiest to decipher because each signal has one specific purpose. All but two of these signals are issued exclusively by the controller.

Uniline Commands

- Interface Clear (IFC) is the most dramatic uniline command. It stops all activities on the bus and returns the interface of every device to a quiescent state.
- Remote Enable (REN) informs the devices on the bus that the IEEE interface is active. It does not lock out an instrument’s front panel.
- Attention (ATN) is the signal that differentiates data from multiline commands on the data bus. When ATN is asserted by the controller, the bits on the data bus are actually multiline commands being issued to all of the devices on the bus.
- End Or Identify (EOI) can be issued by any device talking on the bus. Talkers use EOI to notify listeners that the end of the transmission has taken place.
- Service Request (SRQ) can be issued by any device on the bus. It allows devices on the bus to interrupt, or alert, the controller to an internal situation that needs servicing (e.g. “my buffer is full,” “I’ve encountered an error,” “my trigger has been satisfied”).

Multiline Commands

Now that you have seen the uniline commands and have a sense for their application, we'll discuss multiline commands. Remember, multiline commands are transmitted from the controller to the devices on the data bus. The devices know that they are not data because the ATN line is asserted. When ATN is asserted by the controller all of the devices must listen to the commands.

Multiline commands serve several functions, most notably to address instruments to talk or listen. To address a device to listen, the controller will assert ATN and place the listen address of the selected device on the data bus. There are 31 listen addresses. These are called the Listen Address Group (LAG). A similar process is used for the Talk Address Group (TAG).

Most IEEE drivers, including IOtech's Driver488 for our line of IEEE controllers, have high level commands that perform several elemental IEEE 488 operations. In the IOtech Driver488 manual, every command explanation contains a field called BUS STATES. In BUS STATES, a complete explanation of what is happening on the bus is displayed. For example, let's examine IOtech's Driver488 command ENTER, which simply gets one reading from a specified device:

BUS STATES:

ATN•UNL, MLA, TAG, *ATN, data..., ATN

First, this indicates that ATN is asserted. Next, the multiline command UNListen (UNL) instructs all devices that were in the listen state to exit that state. The controller then issues My Listen Address (MLA), its own address in the listen address group, and issues the Talk Address Group (TAG) for the specified device. Next, it unasserts ATN, which notifies the addressed device that it may now transmit its data. Finally, after the data has been sent (perhaps ending with an EOI), the controller once again asserts ATN.

Analyzing the IEEE Bus

The simplest way to decipher the controller's operations and the response of the instruments, regardless of what software or hardware you are using, is with an IEEE analyzer. Analyzer488 from IOtech allows the programmer to view all of the transactions on the bus in real time or to record them into its 32K non-volatile transaction buffer for later inspection.

The following example problems are all diagnosed using the Analyzer488. Analyzer488 can be operated as a portable bench-top analyzer from its easy to use keypad, or from the included Analyst488 PC software. Analyzer488 allows the events on the IEEE bus to be monitored, stored and analyzed. It can also be used to control devices on the bus for exercising and verifying instrument operation. The Analyzer488 will automatically translate the state of the data bus and control lines into easy to read IEEE messages or ASCII equivalents like SPE, TAG16, CR, and LF. Along with its large capture buffer, Analyzer488 contains a comprehensive set of trigger features that allow the desired group of transactions to be easily pinpointed and identified.

Common Problems and Solutions

Occasionally systems will encounter problems due to the interaction of several devices in the system. These are among the most difficult problems to debug. You should connect an Analyzer488 and let it run while the application is processing. Recording the bus transactions as they occur and inspecting the transactions one at a time will usually allow you to diagnose these types of problems rather quickly.

Often the problems encountered in a system are due to interactions between one device and the controller. Here is a list of common symptoms and their suggested solutions:

"I get a time-out error whenever I try to send device-dependent commands to my instrument."

The first thing you should check is the setting of IEEE addresses. Every device on the bus must have a unique address between 0 and 30. When sending Device-Dependent Commands (DDCs) to an instrument to change its state or operating mode, the device will first be addressed to listen, then the data will be sent. If the device has TALK and LISTEN indicators on its front panel, you can tell immediately if the address used by the controller matches the actual address of the instrument. If the LISTEN indicator does not come on when sending commands to the device, you are probably using the wrong address for that device.

As we mentioned in the tutorial section of this note, when the ATN line is asserted by the controller all of the instruments on the bus will handshake with, and accept data from, the controller. After the time-out is received, step through the transactions recorded by the Analyzer488. If no instrument addressing commands such as Listen Address Group 16 (LAG16) were recorded, your instrument is probably off or broken, or the cable is disconnected. Regardless of the present state of the instrument, it should handshake (accept data) when the ATN line is asserted. If the addressing commands were successfully recorded on the analyzer, step through the transactions until the ATN line is unasserted. If there are no more recorded transactions, then no instrument was placed in the Listen mode. The controller had no one to handshake with so it "timed-out." Your instrument is probably set to the wrong address.

"At certain points in my program, the system stops and I receive a time-out error."

If portions of your program are operating correctly, then you can be certain that your addresses are set correctly. If you encounter a time-out error in your program after other instrument tasks have been completed successfully, you may have encountered an instrument-readiness problem.

IEEE interfaces and software like IOtech's Personal488 operate very rapidly and can sometimes out-run the instrument they are controlling. For most instruments, data requests are performed in two steps: sending the necessary setup or inquiry commands via DDCs, then addressing the device to Talk. It is possible to issue the necessary commands to request the data from the instrument and then address it to Talk long before it is prepared to supply the requested data. Many instruments will simply pause the bus until they have prepared the data to send. However, other instruments react poorly by "hanging up."

To check for this "outracing" condition, place the Analyzer488 into the Slow Handshake mode. This will effectively slow the transaction speed of the bus to a rate set by the Analyzer488. If the data request takes place successfully, it is probably an "outracing" condition.

“My instrument seems unaffected by the commands I send to it.”

If you have already made certain that you are sending the commands to the right instrument address, you may have left off a crucial piece of information that instructs the instrument to process the commands.

IEEE systems usually use data delimiters called terminators. A Talker will inform a Listener that the data string has come to an end by appending a predefined terminator to the end of its data string. Although terminators are issued solely by the talking device, the listening device(s) must know what terminator to expect. Usually IEEE instruments will issue a carriage return (CR) and a line feed (LF) as their terminator. Some instruments will not process the incoming command string until they detect the proper terminator. You should step through the transactions captured by the Analyzer488 to verify the transmission of the terminator, then make certain that it agrees with the terminator expected by your instrument.

Some instruments have a DDC which instructs the instrument to process all of the previously received commands. This EXECUTE command (typically a character like ‘X’) allows a programmer to send several commands to an instrument in any order over any length of time and then execute them all simultaneously within the instrument. If the EXECUTE DDC is not sent, the state of the instrument will not change. It will react as if the commands were never received.

“When I ask for data, nothing is returned.”

This could be an address or terminator problem like the ones discussed above. See the previous sections to diagnose these problems.

Not all instruments are ready to supply data whenever you ask. Some instruments have nothing to say until they are commanded to acquire or generate data. Some data acquisition instruments have triggering features which allow the instrument to collect and transmit data only after a specified event has occurred. A typical multimeter might have a default trigger of TRIGGER ON TALK which would enable the multimeter to take a reading every time the controller addressed it to Talk. If the same multimeter was set to TRIGGER ON GET, no reading would be available until the controller issued a Group Execute Trigger.

If the device has no data to give, the Analyzer488 will show that the controller has been addressed to Listen and the device was addressed to Talk and then the process stopped. The handshake indicators show that Not Ready For Data (NRFD) was unasserted by the controller but the instrument never asserted Data Valid (DAV). Make certain that your device has data to transmit before you ask it for some.

IOtech’s Driver488 has the capability of assigning a time-out value to the system. If an instrument does not respond within the specified time-out, the process is aborted. In some instances, an instrument may be unable to respond within the specified time-out period and the time-out period will have to be increased.

"When I ask for data, bad data is returned."

Many times the variability of data formats of an instrument will cause problems. Devices can transmit data in binary, ASCII, BCD, packed BCD, or anything else that will fit into 8 bits. Data terminators can be EOI, a byte count, or imbedded characters like CR LF. Data can be sent with prefixes, suffixes, or full headers. IOtech's Driver488 can account for all of these parameters, but some other drivers may not allow this level of flexibility.

When using higher level software packages, the problem of data formats may be impossible to overcome. Usually, menu-driven and turnkey packages go to great lengths to hide the IEEE bus from the operator. The documentation, therefore, makes no attempt to inform the operator of what is actually happening on the bus.

You may encounter a problem if your instrument transmits data in a format that is not recognized by your software package. Check your instrument manual for data format characteristics. Does your instrument transmit non-numeric prefixes or suffixes; is the data in binary or ASCII? Some software drivers will automatically throw away any non-numerics. Others do not. Even if your software throws the non-numerics away, you may encounter problems with instruments that transmit numbers like channel tags in their data prefix.

Most instruments, including IOtech's ADC488 analog to digital data acquisition instrument, can be programmed to adjust their data format for software compatibility. Analyzer488 allows you to quickly inspect the data being transmitted by your instrument, enabling you to make the proper adjustments in your software.

"An SRQ from an instrument sometimes causes a catastrophe."

The asynchronous nature of instrument interrupts can sometimes cause elusive problems. The best way to attack a problem like this is to start the Analyzer488 recording and just let it and the system run. Analyzer488 has a large 32K transaction buffer that is configured in a circular fashion. After 32K transactions have been recorded, new transactions will overwrite the oldest transactions. There is a very high probability that the events leading up to the system "crash" will still be in the recorded memory (not overwritten) after the system has locked-up. Stepping backwards in memory can usually uncover the sequence of operations that caused the problem. The Analyzer488 can also be set up to trigger on the occurrence of one or several SRQs with both a post and pre-trigger assigned. In this way a specified number of events can be captured before and after the occurrence of an SRQ. The Analyzer488 also has comprehensive search features allowing the capture buffer to be scanned for all of the occurrences of any event, including an SRQ.

Some instruments have the capability of generating an SRQ for any of several internal events. Usually an SRQ mask is sent to the instrument to instruct it to generate an SRQ for only a selected subset of those events. Some instruments, by default, will interrupt the controller with an SRQ when an internal error is encountered and not respond to any further bus transactions until the interrupt is serviced. The next time your application program requests data from that instrument, your system will fail. By inspecting the Analyzer488 transaction recording working backward from the end, it will be obvious that an SRQ was asserted by someone on the bus and that it remained without service.

"My system occasionally locks up."

This is another of those intermittent problems that can take a long time to troubleshoot, especially if the mean time between failures is several hours, days or months. As before, the best way to approach the problem is to allow the Analyzer488 to record all of the transactions occurring on the bus. When the number of transactions goes beyond 32,767, the capture pointer will wrap around and continue to record. The last 32,767 transactions will always be stored in memory. When the system crashes, the processing of IEEE bus transactions will probably end also. With the last 32K transactions captured in memory, it is easy to step back through the capture buffer and decipher the sequence of operations that caused the crash.

One possible cause for an intermittent crash problem is the asynchronous occurrence of SRQs as discussed above. There may be areas in your application program that do not react well to being interrupted. Since the SRQ can happen at any time, it may or may not occur during the processing of this sensitive area. But the longer the system runs, the probability that the SRQ will happen at exactly the wrong time increases. A sensitive area may be a part of your code that uses a group of closely related variables that are modified by the SRQ handler. For example, three IEEE 488 counters are used to take readings from three motion encoders. Each counter is programmed to generate an SRQ when its count reaches 256. The SRQ handler reads all three counters and stores them into three separate variables used later by the main program. The main program has a loop that reads the three variables, combines them with some calculation, and sends commands to a motor controller. If the main program was in the process of using the variables and an SRQ occurred (which modifies all three variables), the main program may end up using one old value and two new ones in its calculation.

One way to avoid this kind of problem is to disarm the automatic SRQ vectoring during the processing of sensitive program areas. IOtech's Driver488 has several means by which to arm, disarm and synchronize the servicing of SRQs to your program.

Another source of system malfunctions is from the instruments themselves. Most of today's complex instruments are microprocessor controlled. The internal processor handles the collection of data, the changing of programmable states, the monitoring of trigger events, and the communication on the IEEE interface. These instruments are actually computers, prone to all of the same problems as any other computer.

It is possible that your instrument reacts improperly to a perfectly good application program. The transaction report that Analyzer488 prints out can be used to communicate instrument problems to the manufacturer. The report is easy to read and concisely describes the operation of the controller and the response of the instrument.

New Standards Simplify Programming

Many of the difficulties encountered during the development of IEEE software are due to the non-standard elements of operating the IEEE bus. Terminators, common command syntax, and SRQ handling, among others, were not standardized within IEEE 488.1 and were left to the instrument and controller designers to deal with.

New standards and extensions to older standards are now becoming available that may significantly simplify IEEE system integration. Although supported by only a few instruments presently, standards like IEEE 488.2 and SCPI (Standard Commands for Programmable Instruments) are changing the way instruments are controlled.

The IEEE 488.2 standard is, for the most part, an extension to the present standard, IEEE 488.1. There are only a few hardware differences between the new and old standards, assuring that older instruments conforming to IEEE 488.1 can still be used alongside newer IEEE 488.2 instruments.

The major difference between the two standards lies in the software protocol. Previously non-standard elements of bus communication such as terminators and data types have now been included in the standard. These standards will eliminate some of the variables encountered when integrating instrumentation systems, making debugging simpler.

Frequently Asked Personal488 Questions

Why does the driver return the error message "Time Out on Command?"

The most common cause for 'Time Out on Command' error is an improperly installed Personal488 interface. Hardware conflicts, defective interfaces and miss-configured software all lend this type of time out errors.

In addition, failing to power your instruments or failing to connect the IEEE488 cable can also cause this error

What is the meaning of the following error message "Time Out On Data Read?"

The most common cause for this time out error is incorrectly configured EOS terminators. If Personal488 is expecting a Line Feed (LF) when a Carriage Return (CR) is used, then the transmission will never terminate and a time out error will occur. To correct this problem, review your instrument's documentation to determine the correct EOS terminator.

Another cause for this time out error is an incorrectly programmed instrument. If an instrument does not have data to send or is not in the proper mode for transmitting data a time out error will occur. To correct this problem, refer to the instrument's documentation for the correct programming procedures.

What are IEEE488 terminators?

IEEE488 terminators, sometimes referred to as the EOS characters, are special characters used to signal the end of a data transfer. Typical terminating characters are carriage return <CR> and line feed <LF> and problems will occur when terminators are mismatched. For example, if Personal488 sends <CR><LF> when an instrument expects <LF>, an error typically happens.

Why does my program run once then post the error message "Driver Not Loaded?"

This error message is typically encountered within the Visual Basic environment. Within a Visual Basic application, all active device handles must be closed before the application exits and returns to the Visual Basic environment. If any device handles are left, then upon restarting the program Driver488 will report "Driver Not Loaded" indicating that the handle is unavailable. To recover from this error, exit and restart Visual Basic. Restarting Visual Basic will force Driver488 to close all the handles and unload from memory. To avoid this error use the Close function on all the open handles before stopping the application.

When running WinTest, why do I get the message "Driver Not Loaded ?"

The most common cause for this condition is an incorrect installation or incorrectly configured hardware. Refer to the section *"Driver and Support Software Installation"* or *"Installation Verification"*

My program has been running on an older PC for many years without any problems. Now I have installed Personal488 on my new PC and verified that the installation is working correctly. When I run my program it stops and reports the message “Time out on data read.” Why?

With today’s fast PC, timing problems often occur when an application migrates from a slow PC to a fast PC. This is due to programmers sprinkling delays throughout their code in order to wait for instruments to perform tasks. Some programs, originally written on slow PCs, did not use delays because code execution was slow. Some programmers even used For..Next loops as delays. All of these scenarios become problems when faster PCs are used. For example, inherent delays in slow code disappear; For..Next loop delays become shorter; And programs stop running. These issues can be avoided by polling instruments for status before sending or receiving data. If time delays must be used then make sure the delay reads the system clock for timing information.

I am programming in Visual Basic. My interface is installed correctly, but the “Enter” function returns immediately with no data. Why?

The most common trap Visual Basic programmers fall into is not allocating space in a string variable. Dimensioning the string variable is not enough. Adequate space must be allocated in the string for the expected data. For example, MyString = Space$(10) allocates enough space for 10 characters in MyString.

Appendix D

Hardware Specifications

PCI488 Specifications

Note: These specifications are subject to change without notice.

IEEE 488 Controller Device: IOT7210
Maximum Transfer Rate: 32-bit: 1 Mbyte/s (reads and writes)
Dimensions: Half-size board; occupies one PCI slot
IEEE 488 Connector: Accepts standard IEEE 488 connector with metric studs
Digital I/O Connector: Standard 9-pin female DSUB connector
Power: 500 mA max @ 5 V from PC
Environment: 0 to 70°C, 0 to 95% RH, non-condensing
Digital I/O: Each signal can source 2 mA @ 3.7 V (6 mA @ 3.2 V) and
sink 2 mA @ 0.4 V (6 mA @ 0.9 V)
Multiple Boards: Up to four PCI488 boards can be installed into one PC

AT488pnp Specifications

Note: These specifications are subject to change without notice.

IEEE 488 Controller Device: IOT7210
Maximum Transfer Rates: 16-bit DMA: 1 Mbyte/s (reads), 800 Kbyte/s (writes)
Dimensions: Full-size board, two card edges; occupies one ISA slot
IEEE 488 Connector: Accepts standard IEEE 488 connector with metric studs
Digital I/O Connector: Standard 9-pin female DSUB connector
Power: 1.0 A max @ 5 V from PC
Environment: 0 to 70°C, 0 to 95% RH, non-condensing
DMA: 16-bit DMA on channels 5, 6, and 7
Interrupts: IRQ 3, 4, 5, 7, 10, 11, 12, or 15
Digital I/O: Each signal can source 2 mA @ 3.7 V (6 mA @ 3.2 V) and
sink 2 mA @ 0.4 V (6 mA @ 0.9 V)
Multiple Boards: Up to three AT488pnp boards can be installed into one PC

CARD488 Specifications

Note: These specifications are subject to change without notice.

IEEE 488 Controller Device: IOT7210
Maximum Transfer Rate: 1 Mbyte/s (reads and writes)
Dimensions: Type II (5 mm) PCMCIA card
Bus Interface: PCMCIA PC Card Standard 2.1
IEEE 488 Connector: Accepts standard IEEE 488 connector with metric studs via custom cable
Cable: PCMCIA to IEEE 488, CA-137 (included)

AT488 Specifications

Note: These specifications are subject to change without notice.

IEEE 488 Controller Device: IOT7210
Maximum Transfer Rates: 16-bit DMA: 1 Mbyte/s (reads), 800 Kbyte/s (writes); 8-bit DMA: 330 Kbyte/s (reads), 220 Kbyte/s (writes)
Dimensions: Full-size board, two card edges; occupies one ISA slot
IEEE 488 Connector: Accepts standard IEEE 488 connector with metric studs
Power: 1.0 A max @ 5 V from PC
Environment: 0 to 70°C, 0 to 95% RH, non-condensing
DMA: 16-bit DMA on channels 5, 6, and 7; 8-bit DMA on channels 0, 1, 2, and 3 (jumper selectable)
Interrupts: IRQ 2, 3, 4, 5, 6, or 7 (8-bit slot); IRQ 3-7, 9-12, 14, or 15 (16-bit slot)
IEEE 488 I/O Base Address: `&H02E1`, `&H22E1`, `&H42E1`, or `&H62E1`
Multiple Boards: Up to four AT488 boards can be installed, sharing a single DMA channel and interrupt line

GP488B Specifications

Note: These specifications are subject to change without notice.

IEEE 488 Controller Device: IOT7210
Maximum Transfer Rate: 8-bit DMA: 330 Kbyte/s (reads and writes)
Dimensions: Half-size board, one card edge; occupies one ISA slot
IEEE 488 Connector: Accepts standard IEEE 488 connector with metric studs
Power: 650 mA max @ 5 V from PC
Environment: 0 to 70°C, 0 to 95% RH, non-condensing
DMA: 8-bit DMA on channels 0, 1, 2, and 3 (jumper selectable)
Interrupts: IRQ 2, 3, 4, 5, 6, or 7
IEEE 488 I/O Base Address: `&H02E1`, `&H22E1`, `&H42E1`, or `&H62E1`
Multiple Boards: Up to four GP488B boards can be installed, sharing a single DMA channel and interrupt line

GP488B/MM Specifications

Note: (1) GP488B/MM is only compatible with the Ampro PC/104. (2) Only GP488B/MM Revision B is discussed in this manual. (3) Microswitches 6, 7, and 8 on switch SW1 do not have a function on this board. (4) These specifications are subject to change without notice.

IEEE 488 Controller Device: IOT7210
Maximum Transfer Rate: 8-bit DMA: 330 Kbyte/s (reads and writes)
Connector: 26-pin header ribbon cable to IEEE 488 standard connector
Power: 650 mA @ 5 V from PC
Environment: 0 to 70°C; 0 to 95% RH, non-condensing
DMA: 8-bit DMA on channels 0, 1, 2., and 3 (jumper selectable)
Interrupts: IRQ 2, 3, 4, 5, 6, or 7
IEEE 488 Base I/O Addresses: `&H02E1`, `&H22E1`, `&H42E1`, or `&H62E1`

Appendix E

National Instruments — Driver488/NI/32 2.0

Overview

This package is installed automatically by the Windows 95/98/Me/NT/2000 setup program.

Driver488/NI/32 will allow applications developed for use with National Instruments 32-bit IEEE488 interface driver (GPIB-32.dll) to communicate through the IOtech Personal488 line of interfaces. 32-bit Windows programming languages, including LabView for Windows, are supported.

The following Personal488 interfaces are supported:

IOtech Personal488
IOtech Personal488/AT
IOtech Personal488/ATpnp
IOtech Personal488/CARD
IOtech Personal488/MM
IOtech Personal488/PCI

The hardware I/O settings (port, interrupt, etc...) are stored in the registry. The Control Panel Applet takes care of all configuration information.

Program Requirements

The following software is required:

- Microsoft Windows 95/98/Me/NT/2000 Operating Systems
- Driver488/W95 or /WNT
- Application written to communicate through Gpib-32.dll.

Installation

The IOtech hardware and standard 32-bit windows drivers must first be correctly installed and configured for use with Driver488/NI/32.

If it is not, follow these steps:

1. Install Driver488/W95 (or WNT) and the Personal488 hardware as described in the User Manual. You must have a working software and hardware installation before proceeding to the next step. The Driver488/W95 installation copies the Driver488/NI files to your computer in a directory called "Driver488 NI" under "Programming Language Support\Compatibility" under the installation directory.

2. Copy the file Gpib-32.DLL into the Windows System directory (normally "C:\Windows\System") over the existing file. (You may want to make a backup copy of the existing Gpib-32.DLL before performing this step.)

Upgrading from a Previous Version

If you have been using the 16-bit version, communication was through Gpib.dll. The 32-bit version uses Gpib-32.dll. Follow the instructions in step 2.

Miscellaneous Hints and Tips

This driver is a direct replacement of National Instruments GPIB-32.DLL. No program modifications are required. IBDIAG, IBTEST, IBIC, & GPIBINFO are not supported. Asynchronous I/O is not explicitly supported and will be treated as synchronous.

The interface "IEEE0" must be defined in the "IEEE488" control panel applet. The interface hardware selected for "IEEE0" will be used.

The interface name "GPIB0" is inferred and does not have to be defined. Any additional device names (ex. "DEV1") should be defined in the "IEEE488" control panel applet if they are to be explicitly used.

EOI must accompany any termination characters on data reads.

File Structure

(Windows - System directory)\GPIB-32.DLL (driver, installed)