Dynamic Industrial Interface
V 1.8

A Universal
Application Programmers Interface
To
Data Acquisition Products

Users Manual

Design & Implementation by
NT-ware Systemprogrammierungs GmbH

Contents

1. Introduction

This document provides the Dynamic Industrial Interface Specifications, including all function calls, installation requirements, and operating procedures.

Disclaimer:

NT-ware Systemprogrammierungs GmbH (NT-ware) cannot take responsibility for consequential damages caused by using this software. In no event shall NT-ware be liable for any damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising out of the use of or inability to use this product, even if we have been advised of the possibility of such damages.

2. Features

The Dynamic Industrial Interface (DII) was created to provide a standard way to access the functionality provided by all data acquisition products. This driver interface is not limited to PC cards or Pocket I/O adapters but is specifically geared towards providing support for future product generations currently being designed.

Specifically, the DII provides the following features:

- Platform-independent
  The interface is binary-compatible between Windows NT and Windows 95. The binary-compatibility between Windows NT and Windows 95 guarantees that programs written for either operating system will work unchanged on the other, even without recompilation.

- Abstracts Card Functionality from Card Design
  The interface concentrates on a card's functionality and hides the user from having to know specifics about the card design, for example, which port needs to be accessed in order to access specific functionality.
  This also means that the ISA 16 Relay card is to be programmed exactly like the ADDIO 16 Relay output card. All details of the ADDIO implementation, such as parallel port access are hidden from the user.

- Device Naming
  As a new feature, each device can be given it's own name, specified by the user upon device installation. This allows easier access of devices and cards in situations, where multiple cards of the same type are installed. Furthermore, this also makes the portability of applications between products much easier. For example, on one machine, the user might want to use a Pocket I/O module connected to a sensor, on a different machine, he might want to use an ISA card. By giving both devices the same name, he can move his application to different machines without changing any source code.

- Standardized Notification Mechanism
  The library provides a standardized way of handling notifications or interrupts send by Industrial I/O cards. Interrupt handling is normally difficult for users to handle under Windows 95/NT, since interrupt handling requires writing of Kernel-Mode device drivers. However, interrupt based card operation will extremely increase the performance and usability of Industrial I/O cards under Windows 95/NT, as polling will slow down the whole operating system and will become unbearable in most situations.
  By providing a standard way for users to receive notifications whenever an Industrial I/O cards sends an interrupt, users can take full advantage of interrupt driven I/O cards.
  The notification functions of the DII Library are be enhanced by the OCX, as the notifications will be available as OCX events and can simply be handled by applications like Visual Basic.

- Programming Language Independent
  The library provides a language independent way to access the industrial I/O cards, by using a Dynamic-Link-Library architecture.

- Full 32-bit support
  The library fully supports 32-bit applications on Windows NT/95. That means that Visual Basic 4.0/5.0, Delphi, Visual C++, etc. can be used to access I/O Adapters. Utilizing the OCX/ActiveX control, even JAVA applications may become Industrial I/O capable.

- Full multithreading support
  The Dynamic Industrial Interface fully supports multithreading, so that the same application can access hardware concurrently. The OCX component supports the apartment threading model.

- Portability and stability:
  The user will be able to move his application easily from a desktop computer, where he has an ISA ADDA Card to his laptop where he has a Pocket A/D Box, without changing his application, not even recompiling.

3. Distribution Contents

The Dynamic Industrial Interface Library Distribution consists of the following components:

1. Windows NT Kernel Drivers

2. Windows 95 VxD Drivers

3. A Control Panel Applet to edit/view/modify the device configuration.

4. A Win32 DLL for accessing the driver. The driver allows to access the Windows 95 VxD and the Windows NT driver as well.
   In order to access the DLL, a C-Header file and a Visual Basic file containing the function declarations are included.

5. A OCX/ActiveX component.

6. A Visual C++ sample application
   The Sample application demonstrates the use of the driver API by graphically showing the states of input and output ports and allows to set/reset the output ports with the ease of a mouse click.
   Furthermore, the state of analog channels for AD/DA devices, and some testing for 8253 timer chips are shown.

   The application runs unchanged under Windows 95 and Windows NT.

   It was developed using Visual C++ 5.0. The full source code is included.

   After installation, you can find the Visual C++ Demo in the subdirectory “DiiDemo” under your installation directory.

7. A Visual Basic sample application
   The Visual Basic sample is similar to the Visual C++ sample, except it’s written using Visual Basic Version 5.0 and utilizes the OCX/ActiveX component shipped with the DII.

   After installation, you can find the Visual Basic Demo in the subdirectory “DiiDemoVB” under your installation directory.

8. A Delphi Demo application
   The Delphi sample is similar to the Visual C++ sample, except it’s written using Delphi Version 2.0 and utilizes the OCX/ActiveX component shipped with the DII.

   After installation, you can find the Visual Basic Demo in the subdirectory “DiiDemoDelphi” under your installation directory.

4. Installation

Installation of the Windows NT driver files and sample applications:

1. Insert the driver disk into your floppy drive and execute the program „Setup.Exe“ in the root directory. This application will guide you through the setup process.

2. When the installation is finished, you may add devices using the supplied control panel applet. Under Windows NT 4.0, please select „Start/Settings/Control Panel“ and select the Dynamic Industrial Interface icon. Under Windows NT 3.51, please start the control panel from the Program Manager.

3. Using the control panel applet, you may now add devices by first selecting the appropriate product types (PC Cards or Pocket I/O cards)

4. Press the „Add...“ button to add a new device to the list of installed devices. A dialog box with the list of supported devices is being displayed. Please find the device you want to install or a compatible one.

5. For the device type you have selected, you may now modify the port base address and the device name. For more information about the device name, please see the Chapter „Device Naming“.

6. Please repeat steps 3. - 5. for all devices you would like to install. For more detailed information about setting up devices, please refer to Chapter 4.1 and 4.2.

7. You may finish your installation by pressing „Ok“ in the control panel applet.

Installation of the Windows 95 driver files and sample applications:

1. Insert the driver disk into your floppy drive and execute the program „Setup.Exe“ in the root directory. This application will guide you through the setup process.

2. Use the Windows 95 Hardware Wizard located in the Control Panel, to add new devices supported by the DII. Select “Industrial I/O Devices” as the device class to add. The DII device selector will appear.

3. Choose the device you would like to add to your system. Windows 95 will allocate conflict-free I/O ports and IRQ’s for your device. If you would like to change these settings, you have to go into the Device Manager, located in the System Control Panel.

4. After accepting these settings, you will be prompted to enter a name for the device. To find out more about names please see the chapter “Device Naming”.

5. Repeat steps 2.-5. for all devices you would like to add. For more detailed information about setting up devices, please refer to Chapter 4.1 and 4.2.

4.1. Installation / Removal of ISA I/O Cards

For ISA cards, the installation procedure is slightly different for Windows NT and Windows 95, since the DII Device drivers are fully integrated into the Plug And Play Architecture of Windows 95, which is of course not available for Windows NT at this time.

Installation under Windows NT:

1. For Windows NT (3.51 and 4.0), please run the “Industrial I/O Devices” Setup available in your computers control panel.
2. Press the „Add...“ button to add a new device to the list of installed devices. A dialog box with the list of supported devices is being displayed. Please find the device you want to install or a compatible one.
3. For the device type you have selected, you may now modify the port base address and the device name. For more information about the device name, please see the Chapter „Device Naming“. Note that the installation application does not yet detect port conflicts for the I/O Address you have entered.
4. Repeat these steps for every device you want to add to the system.

Removal under Windows NT:

1. For Windows NT (3.51 and 4.0), please run the “Industrial I/O Devices” Setup available in your computers control panel.
2. From the “ISA/PCI I/O Cards” window, select the device you would like to remove.
3. Press the “Remove” button to remove the device from your system.
4. Repeat these steps for every device you want to remove from your system.

Installation under Windows 95

1. From the Windows 95 Control Panel, run the Hardware Wizard.
2. Choose to manually add devices, don’t run the automatic hardware detection
3. Select the “Industrial I/O Devices” device class
4. From the device selection dialog appearing on screen, please choose the I/O card you would like to select and press “OK”.
5. Windows 95 automatically assigns the I/O Base address and optionally an IRQ available for your card. Please make sure the settings on your card match those offered by Windows 95. You can later change the settings assigned by Windows 95 in the Systems control panel.
6. After reviewing your I/O port settings you will be asked to enter a name for the new card or accept the default name created.
7. Repeat these steps for every device you want to add to your system.
8. It is recommended that you reboot your machine after installing any cards.

Removal under Windows 95

1. From the Windows 95 Control Panel, run the System applet, and change to the “Device-Manager” window.
2. You will find all I/O cards you have added under the “Industrial I/O Devices” category. Select the device you would like to remove and press the “Remove” button.
3. Repeat these steps for every device you want to remove from your system.
4. It is recommended that you reboot your machine after removing any cards.

4.2. Installation of Pocket I/O Series Modules

The installation of Pocket I/O modules to your system is identical between Windows 95 and Windows NT.

Installing Pocket I/O Modules

1. For Windows NT and Windows 95, please run the “Industrial I/O Devices” Setup available in your computers control panel.
2. Switch to the “Pocket I/O Modules” page.
3. First, add a port, to which one or more Pocket I/O modules are attached, by pressing “Add Port...”
4. Select, which parallel port you would like to use for connection Pocket I/O, or which serial port you would like to use to communicate with a Pocket COM Module. For the COM Module, also make sure the baud rate and Box-ID settings correspond to the DIP-Switch settings of the COM Module.
5. After adding a port, it appears in the configuration window. Please select the port with your mouse, and press the “Add...” button, to add a Pocket I/O module to the port.
6. Select the type of Pocket I/O module, you would like to add to your system.
7. Enter a name for the Pocket I/O module, through which you can later access the module, or accept the default name created.
8. Adjust the Box-ID to the DIP-Switch settings of your Pocket I/O module. Please refer to your Pocket ADDIO manual for more information
9. Enter the rough length of your cable between your parallel port and your Pocket I/O module. The DII Kernel Drivers use this length to calculate a delay for accessing the devices. The longer the cable is, the longer the delay will be.
10. Repeat these steps for all Pocket I/O modules you would like to add to your system. If you are connecting multiple Pocket I/O modules to the same port, please make sure that the Box-ID’s are different for each box, otherwise you will experience difficulties communicating with all boxes simultaneously.

Please note that you can configure any number of ports with any number of Pocket I/O modules attached to each port, only physically limited by the number of ports your computer has installed, and the number of Pocket I/O modules that are attachable to one port.

Removing Pocket I/O Modules

1. For Windows NT and Windows 95, please run the “Industrial I/O Devices” Setup available in your computers control panel.
2. Switch to the “Parallel Port Pocket I/O Modules” page.
3. If you would like to remove a single device, please select the device with your mouse and press “Remove”.
4. If you would like to remove a whole port with all devices attached to it, please select the port with your mouse and press “Remove Port”.
5. Repeat these steps for all Pocket I/O modules you would like to remove from your system.

5. Deinstallation

When moving to a different machine, for example, the Dynamic Industrial Interface, including all supplied files, registry settings, configuration files, etc. can be easily and automatically be removed.

On Windows 95 and Windows NT 4.0, please use the Control Panel's „Add/Remove Software" applet, and select „Dynamic Industrial Interface". Then press „Remove" and the Dynamic Industrial Interface will be deleted from your machine.

On Windows NT 3.51, please select the „Uninstall Dynamic Industrial Interface" in the Program Manager Group created during the installation.

Please note that also all configuration settings are being deleted. If you would like to use the interface again, you'll have to reinstall the library as mentioned above, and you will have to reinstall every device using the steps mentioned above.

6. Device Naming

The Dynamic Industrial Interface introduces a powerful new feature: Device Naming.

During the installation of devices, you will have noticed, that you can assign a unique name for every device. When creating your application program, you can then access that device using the name you have entered, or you may prompt the user for the name, or even let the user graphically browse for the correct device for your application.

This technique was introduced for the following reasons:

- More clarity when using multiple devices
  Using device names, you can now access more easily and clearly multiple devices of the same type in your machine. For example, if you have 3 8 Relay 8 Photo Isolator cards in your machine, you no longer need to access these cards by their type and some index, which often lead to confusion. You can now just assign a name to each card during the installation, especially a meaningful name.

- More portability
  You will have noticed, one of the goals of the Dynamic Industrial Interface is to make device access more consistent and more clear across all products. The application programmer should not really have to worry about the differences of a Pocket I/O AD module and an ISA card offering AD functionality.
  Furthermore, your software should also be portable between the different products, so you or your end user can also choose the appropriate device at the time your application is being run not the time it is developed.
  Using device naming, for example, you can simply assign a name such as „AnalogIO“ to your product handling Analog Input and Output. On a different machine, you can assign the same name to a different card or product installed. Now, since your software is working with a device called „AnalogIO“ and the Dynamic Industrial Interface hides you from all the details about that device, your software will run unchanged, even on different platforms (Windows NT/ Windows 95).

  Now you can develop software that you can use in-house in your PC, use with Parallel Port Pocket I/O series on your laptop, and take full advantage immediately of future products.

7. Backward compatibility

To ease the transition for users of the previous RelDrv Relay/Photo Isolator driver for Windows NT and Windows 95, the Dynamic Industrial Library has been made compatible with that driver.

By changing the names of the following function calls, you can make your software compatible with the Dynamic Industrial Interface:

RdOpenAdapter	->	DiiOpenDevice (better DiiOpenNamedDevice)
RdCloseAdapter	->	DiiCloseDevice
RdSetSingleRelay	->	DiiSetDigitalBit
RdSetRelaysByteWise	->	DiiSetDigitalByte
RdReadSingleIsolator	->	DiiGetDigitalBit
RdReadIsolatorsByteWise	->	DiiGetDigitalByte
RdReadAnalogChannel	->	DiiGetAnalogChannel
RdWriteAnalogChannel	->	DiiSetAnalogChannel

Other differences:

Under Windows 95, the Dynamic Industrial Interface no longer supports accessing devices without the VxD kernel driver installed.

To reduce the memory overhead for your computer, the Windows 95 VxD is now completely dynamic, it will be loaded when needed, and will be closed when there are no more application using the DII.

Please note that to take full advantage of advanced features like „Device Naming“, you should replace your call to „RdOpenAdater“ with „DiiOpenNamedDevice“.

For further information, please feel free to contact us.

8. Dynamic Industrial Interface function calls

Since the DII was developed in the C++ language, some data types used may not be present in the programming language you want to use.
If you have difficulty using the DLL in your programming language, you may also use the OCX component, which provides easier access from many high-level languages.

Please find the following data type conversion table for your convenience:

HANDLE	An opaque 32-bit integer
BYTE	A 8-bit unsigned integer
BOOL	A 32-bit integer, either 0 (FALSE) or 1 (TRUE)
DWORD	A 32-bit unsigned integer
HWND	A 32-bit integer representing a valid handle to a Window
LPCTSTR	A 32-bit flat pointer to a zero terminated string
LPBOOL	A 32-bit flat pointer to a variable of type BOOL
LPBYTE	A 32-bit flat pointer to a variable of type BYTE
LPDWORD	A 32-bit flat pointer to a variable of type DWORD

Also note that the DLL employs the Standard Call (Pascal) calling mechanism, which is used for all system Dlls as well and is compatible with Visual Basic.

8.1. Functions to open and close Devices

DiiOpenDevice

This function opens a device for further access.

Declaration

```
HANDLE DiiOpenDevice (      DWORD dwDeviceType,
                            DWORD dwIndex,
                            BOOL bExclusive
                        );
```

Parameters

dwDeviceType The type of the device to open. See Remarks for more information

dwIndex The index of the device to open (0 is the first device)

bExclusive Determines, whether to open the device exclusively (TRUE), or allow other applications to use the device, as well.

Return value

A valid handle representing the device, or INVALID_HANDLE_VALUE (-1) if an error occurred.

Remarks

This function is provided for backward compatibility. Please use the new function DiiOpenNamedDevice for new applications.

The *dwDeviceType* may have one of the following values:

RD_SMARTLAB_16CHANNEL	*(1)*
RD_SMARTLAB_8CHANNEL	*(2)*
RD_ICC_8RELAY8ISOLATOR	*(3)*
RD_ICC_16RELAY	*(4)*
RD_ICC_16ISOLATOR	*(5)*
RD_ICC_8SSR8LOGIC	*(6)*
RD_TTL	*(7)*
RD_IBC_32ISOLATOR	*(9)*
RD_ADVANCE_ADDA	*(0x100)*
RD_SUPER_12BIT_ADDA	*(0x101)*
RD_12BIT_ADDA	*(0x102)*
RD_8CHANNEL_DA	*(0x103)*
RD_SUPER_14BIT_ADDA	*(0x104)*

Note: The Windows 95 and Windows NT device drivers are dynamic, that is they are not loaded into the system at boot time. If you attempt to open an adapter for the first time, this might cause the drivers to get loaded and in turn certain Photo-Isolator adapters to get initialized.

DiiOpenNamedDevice

This function opens a device for further access.

Declaration

```
HANDLE DiiOpenNamedDevice (     LPCTSTR lpszDeviceName,
                                BOOL bExclusive
                           );
```

Parameters

lpszDeviceName The name of the device to open. For further information, please see the chapter „Device Naming“

bExclusive Determines, whether to open the device exclusively (TRUE), or allow other applications to use the device, as well.

Return value

A valid handle representing the device, or INVALID_HANDLE_VALUE (-1) if an error occurred.

Example

```
HANDLE hDevice = DiiOpenNamedDevice (“Device*“, TRUE);

if (hDevice == INVALID_HANDLE_VALUE)
{
        MessageBox (NULL,“Open Failed!“,“Error“,MB_OK);
}
```

Remarks

This function is the preferred way of opening a device for later use. The function scans the configuration information provided, and returns a handle to the device, if found.

The *lpszDeviceName* parameter may also include a wildcard character (‘*’). The name provided is then matched against all device names found. The first matching device is returned.
For example: DiiOpenNamedDevice (“Device*“, TRUE); will open a device named “Device0“ or “Device12“, etc.

Note: The Windows 95 and Windows NT device drivers are dynamic, that is they are not loaded into the system at boot time. If you attempt to open an adapter for the first time, this might cause the drivers to get loaded and in turn certain Photo-Isolator adapters to get initialized.

DiiCloseDevice

This function closes a device again.

Declaration

```
BOOL DiiCloseDevice ( HANDLE hDevice );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid.

Example

```
DiiCloseDevice (hDevice);
```

Remarks

8.2. Functions to enumerate or browse devices

DiiSelectDevice

This function displays a dialog box on the screen and allows the user to select one of the installed devices. The name of the device is returned, and can then easily be passed to DiiOpenNamedDevice.

Declaration

```
BOOL DiiSelectDevice (      HWND        hParent,
                            LPTSTR      lpszDeviceName,
                            DWORD       dwMaxDeviceName
                      );
```

Parameters

hParent A valid handle representing the parent window for the dialog box to be displayed or NULL.

lpszDeviceName A pointer to a buffer receiving the name of the device selected.

dwMaxDeviceName The maximum size of the buffer pointed to by lpszDeviceName.

Return value

TRUE if successful and the user has confirmed his/her selection, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_MORE_DATA - The buffer passed by lpszDeviceName was too small.

Example

```
char szDeviceName[201];

if (DiiSelectDevice (NULL, szDeviceName,200))
{
        HANDLE hDevice = DiiOpenNamedDevice (szDeviceName, TRUE);

        DiiCloseDevice (hDevice);
}
```

Remarks

This powerful function allows the programmer to keep his/her application independent of a specific data acquisition device, as it allows to dynamically select a device by the user.

DiiGetInstalledDevice

This function returns a name of a device installed. It can be called multiple times to enumerate all devices installed in a computer.

Declaration

```
BOOL DiiGetInstalledDevice (   DWORD      dwIndex,
                               LPTSTR     lpszDeviceName,
                               DWORD      dwMaxDeviceName
                               );
```

Parameters

dwIndex The index of the device, whose name is to be retrieved.

lpszDeviceName A pointer to a buffer receiving the name of the device selected.

dwMaxDeviceName The maximum size of the buffer pointed to by lpszDeviceName.

Return value

TRUE if successful and the user has confirmed his/her selection, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_MORE_DATA - The buffer passed by lpszDeviceName was too small.

ERROR_NO_MORE_ITEMS - There are no more devices to be returned

Example

```
        //
        // print all devices installed on screen:
        //
char szDeviceName[201];

for (int counter = 0;;counter++)
{
        if (!DiiGetInstalledDevice (counter, szDeviceName, 200))
                break;

        printf ("%d . %s\n",counter,szDeviceName);
}
```

Remarks

8.3. Functions to retrieve information about an Device

DiiGetNumberOfDigitalChannels

This function returns the number of digital channels a device offers (if any)

Declaration

DWORD DiiGetNumberOfDigitalChannels (HANDLE hDevice);

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

Return value

The number of digital channels the device supports

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);
DWORD dwChannels = DiiGetNumberOfDigitalChannels( hDevice );
DiiCloseDevice (hDevice);
```

Remarks

DiiGetNumberOfDigitalInOutChannels

This function returns the number of digital channels a device offers (if any), differentiated by input and output channels.

Declaration

```
BOOL DiiGetNumberOfDigitalInOutChannels (
                                    HANDLE hDevice,
                                    LPDWORD lpdwInputChannels,
                                    LPDWORD lpdwOutputChannels);
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

lpdwInputChannels
A pointer to a DWORD variable receiving the number of input channels for the device.

lpdwOutputChannels
A pointer to a DWORD variable receiving the number of output channels for the device.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the line number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DWORD dwInputChannels;
DWORD dwOutputChannels;

DiiGetNumberOfDigitalChannels( hDevice, &dwInputChannels, &dwOutputChannels );

DiiCloseDevice (hDevice);
```

Remarks

DiiGetNumberOfAnalogChannels

This function returns the number of analog channels a device offers (if any)

Declaration

DWORD DiiGetNumberOfAnalogChannels(HANDLE hDevice);

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

Return value

The number of analog channels the device supports

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);
DWORD dwChannels = DiiGetNumberOfAnalogChannels( hDevice );
DiiCloseDevice (hDevice);
```

Remarks

DiiGetNumberOfAnalogInOutChannels

This function returns the number of analog channels a device offers (if any), differentiated by input and output channels.

Declaration

```
BOOL DiiGetNumberOfAnalogInOutChannels (
                                        HANDLE hDevice,
                                        LPDWORD lpdwInputChannels,
                                        LPDWORD lpdwOutputChannels);
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

lpdwInputChannels
A pointer to a DWORD variable receiving the number of input channels for the device.

lpdwOutputChannels
A pointer to a DWORD variable receiving the number of output channels for the device.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the line number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DWORD dwInputChannels;
DWORD dwOutputChannels;

DiiGetNumberOfAnalogChannels( hDevice, &dwInputChannels, &dwOutputChannels );

DiiCloseDevice (hDevice);
```

Remarks

DiiGetResolution

This function returns the analog resolution a device supports.

Declaration

```
DWORD DiiGetResolution( HANDLE hDevice );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

Return value

A number representing the number of bits available for analog input/output, for example 12 or 14.

If the device does not have analog input/output capabilities, zero is returned.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);
DWORD dwResolution = DiiGetResolution( hDevice );
DiiCloseDevice (hDevice);
```

Remarks

8.4. Functions to for digital input/output

DiiSetDigitalBit

This function sets or clears a single bit on a digital output line.

Declaration

```
BOOL DiiSetDigitalBit ( HANDLE hDevice,
                        DWORD dwLine,
                        BOOL    bState
                        );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwLine The index of the bit on the card to manipulate. The first bit has index 0.

bState The new state of the bit, either set (1/TRUE) or cleared (0/FALSE)

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the line number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSetDigitalBit ( hDevice, 0, 1);          // set's first bit to one.

DiiCloseDevice (hDevice);
```

Remarks

8255 Devices:

This function automatically configures the specified port for output, if the chip was not configured before. If the line specified exists on Port C of a 8255 chip, the chip's bit set/reset feature is used for optimum flexibility.
Since the 8255 chip's lines can be reconfigured for input/output, the bit line index starts counting on the very first line of the first chip, regardless of whether it was configured as input or output at the time the function was called.
The 8255 chip's control port is disregarded for standard input/output. So line 24 will be the first line of Port A of the second 8255 chip on the device (if any)

DiiSetDigitalByte

This function outputs a complete byte to a digital output port of a device.

Declaration

```
BOOL DiiSetDigitalByte( HANDLE hDevice,
                        DWORD dwPort,
                        BYTE   byPortState
                        );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwPort The index of the port on the card to manipulate. The first port has index 0.

byPortState The new state of the port

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the port number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSetDigitalByte( hDevice, 0, 0xFF);          // set's all bits on the first port

DiiCloseDevice (hDevice);
```

Remarks

8255 Devices:

This function automatically configures the specified port for output, if the chip was not configured before. Since the 8255 chip's lines can be reconfigured for input/output, the port index starts counting on the very first line of the first chip, regardless of whether it was configured as input or output at the time the function was called.
The 8255 chip's control port is disregarded for standard input/output index calculation. So port 3 will be Port A of the second 8255 chip on the device (if any)

DiiGetDigitalBit

This function returns the state of a single bit on an input port of a device

Declaration

```
BOOL DiiGetDigitalBit ( HANDLE hDevice,
                        DWORD dwLine,
                        LPBOOL lpbState
                        );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwLine The index of the bit on the card to manipulate. The first bit has index 0.

lpbState A pointer to a variable receiving the new state of the bit, either set (1/TRUE) or cleared (0/FALSE)

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the line number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

BOOL bState;

DiiGetDigitalBit ( hDevice, 0, &bState);          // reads the state of the first bit

DiiCloseDevice (hDevice);
```

Remarks

8255 Devices:

This function automatically configures the specified port for input, if the chip was not configured before.
If the line specified exists on Port C of a 8255 chip, the chip's bit set/reset feature is used for optimum flexibility.
Since the 8255 chip's lines can be reconfigured for input/output, the bit line index starts counting on the very first line of the first chip, regardless of whether it was configured as input or output at the time the function was called.
The 8255 chip's control port is disregarded for standard input/output. So line 24 will be the first line of Port A of the second 8255 chip on the device (if any)

DiiGetDigitalByte

This function input a complete byte from a digital input port of a device.

Declaration

```
BOOL DiiGetDigitalByte( HANDLE hDevice,
                        DWORD dwPort,
                        LPBYTE lpbyPortState
                        );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwPort The index of the port on the card to read. The first port has index 0.

lpbyPortState A pointer to a variable of type BYTE receiving the new state of the port

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the port number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

BYTE byState;
DiiGetDigitalByte( hDevice, 0, &byState);          // reads the state of the first input port

DiiCloseDevice (hDevice);
```

Remarks

8255 Devices:

This function automatically configures the specified port for input, if the chip was not configured before. Since the 8255 chip's lines can be reconfigured for input/output, the port index starts counting on the very first line of the first chip, regardless of whether it was configured as input or output at the time the function was called.
The 8255 chip's control port is disregarded for standard input/output index calculation. So port 3 will be Port A of the second 8255 chip on the device (if any)

DiiGetOutputPort

This function reads the state of an output port, i.e. a bank of relays.

Declaration

```
BOOL DiiGetOutputPort ( HANDLE hDevice,
                        DWORD dwPort,
                        LPBYTE lpbyPortState
                        );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwPort The index of the *output* port on the card to read. The first port has index 0.

lpbyPortState A pointer to a variable of type BYTE receiving the new state of the port

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the port number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSetDigitalByte ( hDevice, 0, 0x12 );

BYTE byState;
DiiGetOutputPort( hDevice, 0, &byState);        // reads the state of the first output port

DiiCloseDevice (hDevice);
```

Remarks

Some hardware adapters don't offer the ability to read the state of an output port from the hardware. In this case, the DII returns the state of the internal port cache used by the kernel drivers on Windows NT and Windows 95. If the device does offer to read the state of an output port, the DII performs a port input operation.

8.5. Functions to configure digital channels

DiiSet8255Config

This function configures a 8255 chip on the device.

Declaration

```
BOOL DiiSet8255Config( HANDLE hDevice,
                       DWORD dwChip,
                       BYTE byConfiguration
                       );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChip The index of the chip on the card to manipulate. The first chip has index 0.

byConfiguration A byte containing the new configuration for the 8255 chip. This byte must conform to the configuration byte specified in the 8255 chip's data sheet.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the chip number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSet8255Config( hDevice, 0, 0x80);     // configures all ports as output, mode 0

DiiCloseDevice (hDevice);
```

Remarks

This function sets the device configuration for a 8255 chip on the card.
After specifically setting the device configuration, the DII library does no longer reconfigure the ports in response to DiiGetDigitalByte/DiiSetDigitalByte calls.

You may also directly use the chip's bit set/reset feature using this call, as it outputs directly to the control port of the specified 8255 chip.

8.6. Functions for analog input / output

DiiSetAnalogChannel

This function sets an analog channel on the device to a specific value. It takes a 32-bit integer value as a RAW value to be written to the channel.

Declaration

```
BOOL DiiSetAnalogChannel(   HANDLE hDevice,
                            DWORD dwChannel,
                            DWORD dwValue
                        );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The index of the channel on the card to manipulate. The first channel has index 0.

dwValue A 32-bit integer containing the value to output. The value is truncated according to the appropriate resolution of the device

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSetAnalogChannel( hDevice, 0,  0);    // sets the first analog channel to zero

DiiCloseDevice (hDevice);
```

Remarks

This function should be used if the application programmer wants to perform a conversion from the voltage output to the raw device value manually. For easier and more advanced setting of an analog channel, use the function *DiiSetRealAnalogChannel*.

DiiSetRealAnalogChannel

This function sets an analog channel on the device to a specific value. It takes a double value in the range supported by the device and/or setup by the user.

Declaration

```
BOOL DiiSetRealAnalogChannel(   HANDLE hDevice,
                                DWORD dwChannel,
                                double dValue
                            );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The index of the channel on the card to manipulate. The first channel has index 0.

dValue A double (floating point) value containing the value to be set. It is converted to a raw device value depending on the range setup for the device and the resolution supported by the device. This conversion is performed automatically by the DII.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSetAnalogChannel( hDevice, 0,  3.5); // sets the first analog channel to 3.5 V

DiiCloseDevice (hDevice);
```

Remarks

Use this function for an easy, reliable and device-independent setting of analog channel values.

DiiGetAnalogChannel

This function sets an analog channel on the device to a specific value. It returns the raw integral value from the card.

Declaration

```
BOOL DiiGetAnalogChannel(   HANDLE hDevice,
                            DWORD dwChannel,
                            LPDWORD lpdwValue
                        );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The index of the channel on the card to read. The first channel has index 0.

lpdwValue A pointer to a 32-bit integer receiving the current value of the analog channel. The value is truncated according to the appropriate resolution of the device

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DWORD dwValue;
DiiGetAnalogChannel( hDevice, 0, &dwValue);   // reads the first analog channel on the device

DiiCloseDevice (hDevice);
```

Remarks

This function returns the raw analog channel value as given from the card. It does not perform any transformation into the value range setup by the user and supported by the device. To get a real value from the card (i.e. a Voltage), use the function DiiGetRealAnalogChannel.

DiiGetRealAnalogChannel

This function sets an analog channel on the device to a specific value. It returns an interpreted value from the card.

Declaration

```
BOOL DiiGetRealAnalogChannel(   HANDLE hDevice,
                                DWORD dwChannel,
                                double * lpdValue
                            );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The index of the channel on the card to read. The first channel has index 0.

lpdValue A pointer to a double (floating point) variable receiving the current value of the analog channel.
This value is interpreted according to the channel value range setup by the user and supported by the card.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

double dValue;
DiiGetRealAnalogChannel( hDevice, 0, &dValue); // reads the first analog channel on the device

                // dValue now contains the value in Volts of the first analog channel.

DiiCloseDevice (hDevice);
```

Remarks

Use this function to easily and reliably return the value from an analog device.

8.7. Functions to configure analog input / output channels

Starting with V1.7.0, the DII supports more elaborate configuration of analog input and output channels. Notable, the DII supports an automatic conversion from the raw values returned from the hardware to meaningful user values, such as Voltages.

Furthermore, the DII can also automatically convert all values to a specified range setup by the user. If, for example, a device is used for measuring currents in the range from 0-20 mA, equating 0-10 V, the DII can directly return the range 0-20 mA, by setting it up in the function DiiSetChannelRange.

Please note that the function DiiSetChannelRange is merely performing a calculation inside the DII, but does not have any effect on the underlying hardware.

Functions that change the configuration of the hardware, with respect to the supported range of values, are DiiSetChannelGain and DiiSetChannelBipolar.

DiiSetChannelGain

This function configured a programmable amplifier/gain on an analog device.

Declaration

```
BOOL DiiSetChannelGain(     HANDLE hDevice,
                            BOOL bInput,
                            DWORD dwChannel,
                            double dGain
                        );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

bInput Specified whether the gain is to be set on an input channel (TRUE) or output channel (FALSE)

dwChannel The index of the channel on the card. The first channel has index 0.

dGain A floating point value indicating the gain to be setup for the channel.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSetChannelGain (hDevice,TRUE,0,10.0);
        // Configures a 10-times amplification for the first channel.
        // the value range will change from 0-10 V to 0-1 V.

DiiCloseDevice (hDevice);
```

Remarks

DiiGetChannelGain

This function returns the current configuration of a programmable amplifier for an analog device

Declaration

```
BOOL DiiGetChannelGain(     HANDLE hDevice,
                            BOOL bInput,
                            DWORD dwChannel,
                            double * lpdGain
                       );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

bInput Specified whether the gain is to be read from an input channel (TRUE) or output channel (FALSE)

dwChannel The index of the channel on the card. The first channel has index 0.

lpdGain A pointer to a floating point value receiving the gain setup on the channel.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

double dCurrentGain;
DiiGetChannelGain (hDevice,TRUE,0,&dCurrentGain);

DiiCloseDevice (hDevice);
```

Remarks

DiiSetChannelBipolar

This function configured a programmable bipolar mode on an analog device.

Declaration

```
BOOL DiiSetChannelBipolar(    HANDLE hDevice,
                              BOOL bInput,
                              DWORD dwChannel,
                              BOOL bIsBipolar
                          );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

bInput Specified whether the channel is an input channel (TRUE) or output channel (FALSE)

dwChannel The index of the channel on the card. The first channel has index 0.

bIsBipolar A boolean indicating that the channel is bipolar (TRUE) or unipolar (FALSE).

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSetChannelBipolar (hDevice,TRUE,0,TRUE);
                    // Configured the first channel for bipolar mode

DiiCloseDevice (hDevice);
```

Remarks

DiiGetChannelBipolar

This function returns the current configuration of a programmable amplifier for an analog device

Declaration

```
BOOL DiiGetChannelBipolar(    HANDLE hDevice,
                              BOOL bInput,
                              DWORD dwChannel,
                              LPBOOL lpbBipolar
                          );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

bInput Specified whether the channel is an input channel (TRUE) or output channel (FALSE)

dwChannel The index of the channel on the card. The first channel has index 0.

lpbBipolar A pointer to a boolean variable indicating whether the channel is configured for bipolar mode or unipolar mode.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

BOOL bIsBipolar;
DiiGetChannelBipolar (hDevice,TRUE,0,&bIsBipolar);

DiiCloseDevice (hDevice);
```

Remarks

DiiSetChannelRange

This function configured a user-defined channel range for the analog device. All values retrieved from DiiGetRealAnalogChannel or setup by DiiSetRealAnalogChannel will be inside this range.

Declaration

```
BOOL DiiSetChannelRange (   HANDLE hDevice,
                            BOOL bInput,
                            DWORD dwChannel,
                            double dMinimum
                            double dMaximum
                        );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

bInput Specified whether the channel is an input channel (TRUE) or output channel (FALSE)

dwChannel The index of the channel on the card. The first channel has index 0.

dMinimum Specifies the minimum value of the channel

dMaximum Specifies the maximum value of the channel

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSetChannelRange (hDevice,TRUE,0,0,0.020);
                    // changes the returned value range to for example 0-20 mA.

DiiCloseDevice (hDevice);
```

Remarks

With this function, a conversion filter inside the DII is setup. The DII now automatically converts all values to and from this channel from the user-specified range to the range supported by the device. For example, the range can be changed from 0-10 V to 0-0.020 A.

DiiGetChannelRange

This function returns a user-defined channel range for the analog device. All values retrieved from DiiGetRealAnalogChannel or setup by DiiSetRealAnalogChannel will be inside this range.

Declaration

```
BOOL DiiGetChannelRange (   HANDLE hDevice,
                            BOOL bInput,
                            DWORD dwChannel,
                            double * lpdMinimum
                            double * lpdMaximum
                        );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

bInput Specified whether the channel is an input channel (TRUE) or output channel (FALSE)

dwChannel The index of the channel on the card. The first channel has index 0.

dMinimum A variable receiving the minimum value of the channel

dMaximum A variable receiving the maximum value of the channel

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

double dMinRange;
double dMaxRange;
DiiGetChannelRange (hDevice,TRUE,0,&dMinRange,&dMaxRange);

DiiCloseDevice (hDevice);
```

Remarks

DiiGetDeviceChannelRange

This function returns the device-dependent value range. It takes into account the physical hardware limitations and the current configuration of amplifiers.

Declaration

```
BOOL DiiGetDeviceChannelRange (     HANDLE hDevice,
                                    BOOL bInput,
                                    DWORD dwChannel,
                                    double * lpdMinimum
                                    double * lpdMaximum
                               );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

bInput Specified whether the channel is an input channel (TRUE) or output channel (FALSE)

dwChannel The index of the channel on the card. The first channel has index 0.

dMinimum A variable receiving the minimum value of the channel

dMaximum A variable receiving the maximum value of the channel

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel number was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

double dMinRange;
double dMaxRange;
DiiGetDeviceChannelRange (hDevice,TRUE,0,&dMinRange,&dMaxRange);

DiiCloseDevice (hDevice);
```

Remarks

This function is used to return the channel range of the device. It cannot be changed by the user, except by adjusting the gain/amplification for a channel. Most devices support a channel range from 0-10 V without amplification. For devices that support programmable amplification, this range can change.

8.8. Functions to access timers on cards

DiiSetTimerConfig

This function configures timer chips on the device.

Declaration

```
BOOL DiiSetTimerConfig(     HANDLE hDevice,
                            DWORD dwTimer,
                            DWORD dwConfig
                        );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwTimer The index of the timer on the card to access. The first timer has index 0.

dwConfig A 32-bit integer containing the configuration for the timer. On 8253 timers, only the lowest byte is being used.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the timer index was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSetTimerConfig( hDevice, 0, 0x6);     // set square wave generation for timer

DiiCloseDevice (hDevice);
```

Remarks

8253 Chips:
This function sets the configuration byte for a timer on the 8253 chip. The configuration is not actually output, until DiiLoadTimer is called.

DiiLoadTimer

This function loads a value into a timer.

Declaration

```
BOOL DiiLoadTimer(      HANDLE hDevice,
                        DWORD dwTimer,
                        DWORD dwValue
                  );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwTimer The index of the timer on the card to access. The first timer has index 0.

dwValue A 32-bit integer containing the timer value to write.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the timer index was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiLoadTimer( hDevice, 0, 40000);       // load the value into the first timer

DiiCloseDevice (hDevice);
```

Remarks

8253 Chips:
This function actually writes out the timer configuration and loads the timer according to the 8253 specifications.

DiiGetTimer

This function loads a value into a timer.

Declaration

```
BOOL DiiGetTimer(       HANDLE hDevice,
                        DWORD dwTimer,
                        LPDWORD lpdwValue
                );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwTimer The index of the timer on the card to access. The first timer has index 0.

lpdwValue A pointer to a 32-bit integer receiving the current timer value.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the timer index was out of range for the device selected.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DWORD dwValue;
DiiGetTimer( hDevice, 0, &dwValue);     // read the current value of the timer

DiiCloseDevice (hDevice);
```

Remarks

8253 Chips:
A counter latching operation is actually performed, so reading the timer does not interfere with the actual timing.

8.9. Functions to handle standard notification

The following structures and constants are used for the notification mechanism. Please note that these may be specific to the C language. Other language may use the OCX for easier access to notification functions. In the OCX, notifications are implemented as standard OCX events for simplified use.

Note: Notifications as well as OCX Events are only supported for I/O devices that support hardware interrupts. We plan on providing notification support for older cards, as well, at a later time. We recommend that you check the boolean return value of **DiiRequestNotification** to make sure the device supports
notifications.

The following define the notification types that can be requested. Multiple notifications may be combined by a bitwise OR

```
DII_TIMER_EXPIRED 0x000000001
        //              Can be called, when a timer has expired
DII_SIGNAL_CHANGED 0x00000002
        //              Can be called, whenever signals on the card have changed
DII_DATA_BLOCK_READ 0x00000004
        //              Can be called, when background data is available
DII_DATA_BLOCK_SENT 0x00000008
        //              Can be called, as soon as a data block has been sent
DII_VALUE_OUT_OF_RANGE 0x00000010
        //              (reserved for future use)
DII_COUNTER_ZERO 0x00000020
        //              Can be called when a down-counter has reached zero



        //
        // The following data structure is send along with a notification
        // to the user application:
        //

typedef struct _tagDII_NOTIFICATION_DATA
{
        HANDLE      hDevice;            // Identifies the device, which has issued the notification

        DWORD       dwNotificationMask; // A bitmask identifying the notification
                                                    // request this data is for.

        DWORD       dwNotificationParam;
                                        // a 32-bit parameter passed along with the
                                        // notification.
                                        // This might contain a bitmask for the signals that have
                                        // hanged
                                        // or contain the counter index that has reached zero, etc.

        LPVOID      lpBuffer;           // The buffer containing event data, or NULL
        DWORD       dwBufferSize;       // The size of the buffer pointed to by lpBuffer

        DWORD       dwUserData;         // a 32-bit integer for user data (passed into
                                        // DiiRequestNotification

        DWORD       dwAllocationScheme; // Internal: Allocation scheme used for this
                                        // notification data block. DO NOT MODIFY!
```

} DII_NOTIFICATION_DATA, * LPDII_NOTIFICATION_DATA;

typedef void (*PDII_NOTIFY_CALLBACK)(LPDII_NOTIFICATION_DATA);

A table indicating the parameter interpretation depending on the notification type:

Notification Type:	dwNotificationParam	lpBuffer
DII_TIMER_EXPIRED	A zero-based index of the timer that has expired or reached zero.	Not used.
DII_SIGNAL_CHANGED	A bitmask containing all signals that have changed. Currently limited to 32 signals.	Not used.
DII_DATA_BLOCK_READ	The zero-based index of the channel from which the data was read	The actual data being read from the channel. The data is packed into bytes and must be extracted by the user application.
DII_DATA_BLOCK_SENT	The zero-based index of the channel channel to which the data has been written	A pointer to the buffer that was previously send to the DII via DiiOutputBlock
DII_VALUE_OUT_OF_RANGE	<not implemented yet>	<not implemented yet>
DII_COUNTER_ZERO	A zero-based index of the counter, that has counted down to zero.	Not used.

DiiSetNotificationMethodHwnd

This function registers a windows message with the notification mechanism

Declaration

```
BOOL DiiSetNotificationMethodHwnd (  HANDLE hDevice,
                                     HWND hWnd,
                                     UINT    msg,
                                     WPARAM wParam
                                     );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

hWnd A handle to a window to which notification messages will be sent.

msg The message to be sent. The WPARAM parameter of the message will be given, the LPARAM parameter of the message will point to a DII_NOTIFICATION_DATA structure containing more information about the notification

wParam The parameter passed through with the message

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - One of the handles passed were invalid, or the device generally does not support notification

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSetNotificationMethodHwnd (hDevice,
                              AfxGetMainWnd()->m_hWnd,
                                    WM_USER+1,
                              0);

DiiCloseDevice (hDevice);
```

Remarks
This function registers a message, that will be sent, whenever the device needs to notify the user application of some event. Before an application can actually receive events, the function **DiiRequestNotification** needs to be called after this function. This function merely sets the way how to send notifications.

Note: The kernel drivers provide an own intelligent buffering scheme for reading background data. You *must* release the memory from the DII_NOTIFICATION_DATA structure via a call to **DiiReleaseNotificationData** after processing the Windows message. Note that there will be a delay between the notification and the reception of the windows messages, as the messages is not sent synchronously, but rather asynchronously posted to the target window.

DiiSetNotificationMethodCallBack

This function registers a callback routine with the notification mechanism

Declaration

```
BOOL DiiSetNotificationMethodCallBack(      HANDLE hDevice,
                                            PDII_NOTIFY_CALLBACK pfnCallBack
                                            );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

pfnCallBack A pointer to a function that will be called whenever there is a notification outstanding.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - One of the handles passed were invalid, or the device generally does not support notification

Example

```
        // Forward declaration...
void ProcessDeviceNotification (LPDII_NOTIFICATION_DATA);

...

HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSetNotificationMethodCallBack (hDevice,
                                  &ProcessDeviceNotification
                                  );

DiiCloseDevice (hDevice);
```

Remarks

This function registers a callback routine that will be called whenever the device needs to notify the user application of some event. Before an application can actually receive events, the function **DiiRequestNotification** needs to be called after this function. This function merely sets the way how to send notifications.

Note: The function will be called in a different thread, so you will have to employ synchronization methods to synchronize execution with your main code.

DiiRequestNotification

This function requests notifications from the device for specific events that may happen.

Declaration

```
BOOL DiiRequestNotification    (    HANDLE hDevice,
                                    DWORD dwNotificationMask,
                                    DWORD dwUserValue
                                    );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwNotificationMask
A 32-bit integer that specifies the notifications that are requested from the device

dwUserValue A 32-bit integer that gets passed back to the user inside the DII_NOTIFICATION_DATA structure

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - One of the handles passed were invalid, or the device generally does not support any of the requested notifications

Example

```
        // Forward declaration...
void ProcessDeviceNotification (LPDII_NOTIFICATION_DATA);

...

HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

DiiSetNotificationMethodCallBack (hDevice,
                                  &ProcessDeviceNotification
                                  );

DiiRequestNotification (hDevice, DII_SIGNAL_CHANGED, 0);

DiiCloseDevice (hDevice);
```

Remarks

This function actually activates the notification mechanism. Note that before calling this function, either **DiiSetNotificationMethodHwnd** or **DiiSetNotificationMethodCallBack** have to be called to tell the DII how to send notifications back to the user application.

DiiCancelNotification

This function cancels previously requested notifications again. No more notifications for the events canceled will be sent to the user application.

Declaration

```
BOOL DiiCancelNotification      (       HANDLE hDevice,
                                        DWORD dwNotificationMask
                                        );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwNotificationMask
A 32-bit integer that specifies the notifications that should be canceled from the device (Currently ignored)

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid

Example

```
        // Forward declaration...
void ProcessDeviceNotification (LPDII_NOTIFICATION_DATA);

...

HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

if (!DiiSetNotificationMethodCallBack (hDevice,
                                       &ProcessDeviceNotification
                                       )) return;       // device doesn't support notifications

if (!DiiRequestNotification (hDevice, DII_SIGNAL_CHANGED, 0)) return;
                // Request a notification, whenever a signal has changed
getch();        // wait until a key has been pressed, notifications are received in the background.
DiiCancelNotification (hDevice, 0);     // simply cancel everything

DiiCloseDevice (hDevice);
```

Remarks

This function actually activates the notification mechanism. Note that before calling this function, either **DiiSetNotificationMethodHwnd** or **DiiSetNotificationMethodCallBack** have to be called to tell the DII how to send notifications back to the user application.

Note: The *dwNotificationMask* parameter is currently ignored. Calling **DiiCancelNotification** will always cancel all previously requested notifications.

DiiReleaseNotificationData

This function needs to be called to when notifications are sent to a window message, to release the notification structure given. Calling it on the notification data structure passed into the DII_NOTIFY_CALLBACK function has no effect.

Declaration

```
BOOL DiiReleaseNotificationData (       HANDLE hDevice,
                                        LPDII_NOTIFICATION_DATA lpNotifData
                                );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

lpNotifData A 32-bit pointer to the DII_NOTIFICATION_DATA structure that was received via a message.

Return value

TRUE if successful, FALSE otherwise.

Example

```
// Assuming a message handler in the MFC:

void CSampleClass::OnMessage (WPARAM,LPARAM lParam)
{
        LPDII_NOTIFICATION_DATA lpNotifData = LPDII_NOTIFICATION_DATA (lParam);

        // do something with the received notification

        // free the notification structure:
        DiiReleaseNotificationData (    lpNotifData->hDevice,
                                        lpNotifData );
}
```

Remarks

This function is called to release the notification structure passed in via a Windows message. It is necessary only when requesting notifications via Windows messages, since these messages are processed asynchronously.

Note: Failure to call this function will result in constantly increasing memory usage by the DII dynamic link library, as well as potential data loss, since buffers cannot be reused.

8.10. Functions to access kernel mode pulse counters

The Pulse Counter Interface was created to satisfy the demand of detecting digital pulses from high-level languages. Previously, these pulses could only be detected by either polling the hardware I/O ports directly from a high-level language, or, if the device supports interrupts, processing these interrupts within the high-level languages.
Both approaches have disadvantages, the polling is not reliably on multi-tasking operating systems such as Windows NT and Windows 95, and processing interrupts in a high-level language also causes a lot of overhead.
Theoretically, you could count pulses by requesting notifications of type “DII_SIGNAL_CHANGED” from the DII, which will cause a notification to be sent whenever a signal has changed on an interrupt driven I/O card. Note though, that this way of counting pulses has a definite overhead and slow response time. It’s quite likely to loose pulses.

To overcome these problems, the DII now implements a pulse counter interface, both for older I/O cards and newer interrupt driven cards. For older cards, the DII dynamic link library itself performs the polling, which is still better than doing it in a higher-level language such as Visual Basic, but which is still subject to the operating-system scheduling. For interrupt driven cards, pulse counting is done directly in the kernel drivers for Windows 95 and Windows NT.

The Pulse Counting Interface offers the following functionality:

1. Every channel on an digital I/O card can individually be configured as an up or down counter, can individually be reset and preset at any given time.
2. Along with the actual number of pulses, the minimum and maximum time between two pulses is also returned. Note that this information is subject to the resolution of your hardware platform, which may vary.
3. For interrupt driven cards, a “DII_COUNTER_ZERO” notification can be sent, when a down-counter reaches zero.
4. Down-counters can individually be configured to automatically preset themselves when the counter has reached zero (thus to keep counting), or to stay at zero until manually reset again.

The Pulse Counting interface is implemented with the following functions:

DiiEnablePulseCounting

This function enables pulse counting for the card specified.

Declaration

```
BOOL DiiEnablePulseCounting (   HANDLE hDevice,
                                DWORD dwStartPort,
                                DWORD dwNumberOfPorts,
                                DWORD dwReserved
                                );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwStartPort The start I/O port, which shall be monitored. To monitor all ports, specify 0. This parameter is ignored, if the card supports interrupts.

dwNumberOfPorts
The number of ports to support or scan. To monitor all ports, specify 0. This parameter is ignored, if the card supports interrupts.

dwReserved Reserved for future use, specify 0.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

if (!DiiEnablePulseCounting (hDevice,          // enable pulse counting on all ports
                             0,                // return, if pulse counting is not supported
                             0,
                             0)) return;

getch();                    // wait for keypress, pulse counting is done in the background

DiiDisablePulseCounting (hDevice);     // disable pulse counting again.

DiiCloseDevice (hDevice);
```

Remarks

This function attemptys to start pulse counting on the ports specified. For legacy devices, pulse counting is done with 10Hz. For interrupt driven devices, the device is placed into interrupt mode. By default, all counters are configured as up-counters.

DiiDisablePulseCounting

This function disables pulse counting for the card specified.

Declaration

```
BOOL DiiDisablePulseCounting (      HANDLE hDevice
                                    );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

if (!DiiEnablePulseCounting (hDevice,          // enable pulse counting on all ports
                             0,                // return, if pulse counting is not supported
                             0,
                             0)) return;

getch();                    // wait for keypress, pulse counting is done in the background

DiiDisablePulseCounting (hDevice);      // disable pulse counting again.

DiiCloseDevice (hDevice);
```

Remarks

DiiSetDownPulseCounter

This function is called to reconfigure a counter into up-or down pulse counter

Declaration

```
BOOL DiiSetDownPulseCounter     (     HANDLE hDevice,
                                      DWORD dwChannel,
                                      DWORD dwInitialValue,
                                      BOOL bAutoReset
                                );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The channel to reconfigure. The channel corresponds to a single digital I/O line. The first I/O line is specified with 0.

dwInitialValue The initial value for the counter. If non-zero, the counter is configured to be a down-counter, if zero is specified, the counter is configured as an up-counter. The counter is initialized to the this value.

bAutoReset Specifies whether an down-counter shall automatically reset itself (to the initial value specified), when it counts down to zero. When the counter is configured as an up-counter (dwInitialValue is zero), this parameter is ignored.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
HANDLE hDevice = DiiOpenNamedDevice ("*",TRUE);

if (!DiiEnablePulseCounting (hDevice,          // enable pulse counting on all ports
                             0,                // return, if pulse counting is not supported
                             0,
                             0)) return;

DiiSetDownPulseCounter (hDevice,0,1000,TRUE);
                // configure the first line to count from 1000 down, and automatically reset.
getch();                // wait for keypress, pulse counting is done in the background

DiiDisablePulseCounting (hDevice);      // disable pulse counting again.

DiiCloseDevice (hDevice);
```

Remarks

DiiGetPulseCounterValue

This function is called to reconfigure a counter into up-or down pulse counter

Declaration

```
BOOL DiiGetPulseCounterValue    (   HANDLE hDevice,
                                    DWORD dwChannel,
                                    LPDWORD lpdwValue,
                                    LPDWORD lpdwMinTimeBetweenPulses,
                                    LPDWORD lpdwMaxTimeBetweenPulses,
                                    BOOL bResetCounter
                                );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The channel to reconfigure. The channel corresponds to a single digital I/O line. The first I/O line is specified with 0.

lpdwValue A pointer to a variable receiving the current contents of the counter.

lpdwMinTimeBetweenPulses
A pointer to a variable receiving the minimum time between two pulses. The time is given in *microseconds*, although the actual timer resolution of computer will be lower.

lpdwMaxTimeBetweenPulses
A pointer to a variable receiving the maximum time between two pulses. The time is given in *microseconds*, although the actual timer resolution of computer will be lower.

bResetCounter Specifies whether the counter shall be reset while reading. If reset, an up-counter is set to zero, and a down-counter is set to the inital value again. If this parameter is FALSE, the counter is not reset.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
DWORD dwValue;
DWORD dwMinTimeBetweenPulses;
DWORD dwMaxTimeBetweenPulses;

if (!DiiGetPulseCounterValue (  hDevice, 0,
                                &dwValue, &dwMinTimeBetweenPulses,
                                &dwMaxTimeBetweenPulses,
                                TRUE)) return;
    // read a counter and reset it.
```

Remarks

8.11. Functions for continuous background data acquisition

In addition to the standard polling-based mechanism of reading analog or digital values or writing these values to the card, the DII also supports a block based mechanism to read or write large amounts of data samples to and from data acquisition devices. The DII automatically optimizes buffer handling internally, such that the acquisition is continuous. The DII automatically allocates and deallocates all buffers that are required to provide a continuous stream of input and output data to/from the application.

The DII abstracts the hardware design by automatically supporting the most efficient way to configure the underlying hardware device. Some devices may support DMA transfer and interrupt modes, other devices may need an internal polling by the DII.

The background acquisition interface works closely related with the standard notification interface in the DII (explained above), to inform the user application of buffers being received. This notification is asynchronous, either through Active X events, or through function callbacks.

The acquisition interface works for both digital and analog values. Only one type is being supported per acquisition device. In case of reading analog channels, the data returned are raw channel values, either 16 or 32-bit, depending on the resolution of the device.
The DII supports two trigger mechanism, an internal via a timer device on the hardware, or an external, through an external input line.

A prominent feature of the background data acquisition interface is the fact that the sample rate & internal buffer sizes can be chosen per channel. This functionality is not provided with all hardware though, only those cards which offer per-channel configuration of these parameters.

In case the hardware device is acquiring data faster than the user application can evaluate it, the DII supports a streaming to disk, such that data can be streamed to disk first, then later read back into memory via the same interface.

The background acquisition interfaces basically offers 2 variables with which the user can configure the data rate at which his application can accept data. For each channel the *frequency* and the *buffer size* can be configured. The buffer size is always given in bytes. To calculate the number of buffers the application will receive from a specific channel per second, proceed the following way:

*Number Of Data Samples per Second = ChannelFrequency / (BufferSize * BytesPerSample)*

Therefore, the number of data blocks that are passed from the Dii Block Data Acquisition interface to the end user application are dependent on the frequency at which data samples are acquired, divided by the size of the data buffer for each channel. Note that this size is given in bytes, so you have to confirm the number of bytes that are taken per sample. For digital I/O, a data sample is usually 8-bit wide. For analog I/O, the data size depends on the resolution of the device. For cards offering less or equal than 16-bit resolution, the data size is 2-bytes per sample, for higher resolutions, it’s 4-bytes per sample.

DiiPrepareBlockInput

This function is called to prepare a specific channel for block-wise input.

Declaration

```
BOOL DiiPrepareBlockInput   (       HANDLE hDevice,
                                    DWORD dwChannel,
                                    DWORD dwTriggerSource,
                                    DWORD dwTriggerValue,
                                    DWORD dwSampleBufferSize,
                                    LPCTSTR lpszFileName
                            );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The index of the input channel to be scanned automatically (0 is the first channel)

dwTriggerSource Contains the trigger source for the acquisition.

Specify 0 – for a software emulated trigger,
Specify 1 – for a trigger based on an on-board timer
Specify 2 – for a trigger based on an external source, connected via a digital input channel

dwTriggerValue Contains either the frequency in acquisitions per second if an internal timer is used, or the channel number if an external signal is used for triggering an acquisition step.

dwSampleBufferSize Specify the size of the buffers to return for this channel. This is the binary size of the data buffer, not the number of data samples. For analog I/O, usually 1 sample consumes 2 bytes, so specify a size of 2000 to receive 1000 samples per buffer.

lpszFileName An optional filename to which the data is written to allow capturing data directly to disk.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
        // Forward declaration...
void ProcessDeviceNotification (LPDII_NOTIFICATION_DATA);


if (!DiiSetNotificationMethodCallBack (hDevice,
                                       &ProcessDeviceNotification
                                       ))
{
        return FALSE;
}
```

```
if (!DiiRequestNotification (hDevice, DII_DATA_BLOCK_READ, 0))
{
        // could not request notification about data blocks being read
        return FALSE;
}

if (!DiiPrepareBlockInput (     hDevice,
                                0,              // sample the first channel
                                0,              // use internal timer as pacer
                                30000,          // try to sample 30 kHz
                                60000,          // send 30000 AD samples per block
                                NULL            // no file I/O
                ))
{
        // Could not initialize the channel for block input.
        return FALSE;
}

if (!DiiPrepareBlockInput (     hDevice,
                                1,              // sample the second channel
                                0,              // use internal timer as pacer
                                10000,          // try to sample 10 kHz
                                10000,          // send 5000 AD samples per block
                                NULL            // no file I/O
                ))
{
        // Could not initialize the channel for block input.
        return FALSE;
}
```

Remarks

Use this function to configure a single channel for block input. This function can called repeatedly for other channels, before calling DiiStartBlockInput

DiiStartBlockInput

This function is called to start the previously configured block input.

Declaration

```
BOOL DiiStartBlockInput     (       HANDLE hDevice,
                                    );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
        // Forward declaration...
void ProcessDeviceNotification (LPDII_NOTIFICATION_DATA);


if (!DiiSetNotificationMethodCallBack (hDevice,
                                       &ProcessDeviceNotification
                                       ))
{
        return FALSE;
}

if (!DiiRequestNotification (hDevice, DII_DATA_BLOCK_READ, 0))
{
        '' could not request notification about data blocks being read
        return FALSE;
}

if (!DiiPrepareBlockInput (     hDevice,
                                0,              // sample the first channel
                                0,              // use internal timer as pacer
                                30000,          // try to sample 30 kHz
                                60000,          // send 30000 AD samples per block
                                NULL            // no file I/O
                        ))
{
        '' Could not initialize the channel for block input.
        return FALSE;
}

if (!DiiPrepareBlockInput (     hDevice,
```

```
                                        1,              // sample the second channel
                                        0,              // use internal timer as pacer
                                        10000,          // try to sample 10 kHz
                                        10000,          // send 5000 AD samples per block
                                        NULL            // no file I/O
                        ))
{
        ‘’ Could not initialize the channel for block input.
        return FALSE;
}

if (!DiiStartBlockInput    (        hDevice))
{
        ‘’ cannot start block input
        return FALSE;
}
```

Remarks

This function should be called after the individual channels are prepared for block input.

DiiEndBlockInput

This function is called to stop the block based input of data samples

Declaration

```
BOOL DiiEndBlockInput       (       HANDLE hDevice,
                                    );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
        // Forward declaration...
    void ProcessDeviceNotification (LPDII_NOTIFICATION_DATA);


    if (!DiiSetNotificationMethodCallBack (hDevice,
                                    &ProcessDeviceNotification
                                    ))
    {
        return FALSE;
    }

    if (!DiiRequestNotification (hDevice, DII_DATA_BLOCK_READ, 0))
    {
        '' could not request notification about data blocks being read
        return FALSE;
    }

    if (!DiiPrepareBlockInput (     hDevice,
                                    0,              // sample the first channel
                                    0,              // use internal timer as pacer
                                    30000,          // try to sample 30 kHz
                                    60000,          // send 30000 AD samples per block
                                    NULL            // no file I/O
                    ))
    {
        '' Could not initialize the channel for block input.
        return FALSE;
    }

    if (!DiiPrepareBlockInput (     hDevice,
                                    1,              // sample the second channel
                                    0,              // use internal timer as pacer
```

```
                                    10000,          // try to sample 10 kHz
                                    10000,          // send 5000 AD samples per block
                                    NULL            // no file I/O
                        ))
{
        '' Could not initialize the channel for block input.
        return FALSE;
}

if (!DiiStartBlockInput    (        hDevice))
{
        '' cannot start block input
        return FALSE;
}

        //
...     // do some work, i.e. processing the device notifications

if (!DiiEndBlockInput (             hDevice))
{
        return FALSE;
}
```

Remarks

Use this function to stop a background data acquisition in process

DiiGetBlockInputStatus

This function returns status information about the block input for a specific channel

Declaration

```
BOOL DIIEXPORT DiiGetBlockInputStatus    (
                                         HANDLE hDevice,
                                         DWORD dwChannel,
                                         LPDWORD lpdwTotalSamples,
                                         LPDWORD lpdwSamplesPerSecond,
                                         LPDWORD lpdwBufferUnderruns
                                    );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The channel for which information is to be retrieved

lpdwTotalSamples
The total number of data samples (*not bytes!*) being read so far.

lpdwSamplesPerSecond
The average number of data samples per second.

lpdwBufferUnderruns
The number of buffer underruns encountered so far. Buffer underruns occur when the hardware is acquiring data at a faster rate than the DII can process them. This is an error.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
        //
        // see above example about how to setup & start block input

    DWORD dwTotalSamples;
    DWORD dwSamplesPerSecond;
    DWORD dwBufferUnderruns;

    if (DiiGetBlockInputStatus (        hDevice,
                                        0,          // retrieve info for the first channel
                                        &dwTotalSamples,
                                        &dwSamplesPerSecond,
                                        &dwBufferUnderruns
                                ))
    {
        // evaluate the statistical information
    }
```

Remarks

This function returns statistical information during a running block input. The user application can observe that all possible data has been correctly acquired by evaluating the buffer underruns error. A buffer underrun occurs if the hardware sends data at a faster rate than the DII can process and forward it to the user application, or if the user application is not processing the data by the DII fast enough so the DII runs out of buffers.

DiiOpenBlockInputFile

This function is called to re-open a previously created block input file.

Declaration

```
BOOL DiiOpenBlockInputFile    (       HANDLE hDevice,
                                      DWORD dwChannel,
                                      LPCTSTR lpszFileName
                              );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The channel for which the file is opened

lpszFileName The filename of the file.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
BYTE byaDataBuffer[4096];
DWORD dwDataRead;

if (!DiiOpenBlockInputFile      (hDevice,
                                0,
                                “c:\\temp\\Channel0Dump.Bin”))
        return;         // unable to open file

if (DiiReadBlockInputFile       (hDevice,
                                0,
                                byaDataBuffer,
                                4096,
                                &dwDataRead))
        {       // succeeded, so something with the data        }

DiiCloseBlockInputFile          (hDevice,0);
```

Remarks

This function opens a file that was previously created by the block input function. The streaming to disk should be used for data acquisition, if the amount of data acquired cannot be simultaneously evaluated and processed by the application. The application can then first store the data to the disk at a high sampling rate, and later read it back through the block input functions to process it without the time restrictions of online data acquisition.

DiiReadBlockInputFile

This function is called to read data from an opened block input file.

Declaration

```
BOOL DiiReadBlockInputFile    (       HANDLE hDevice,
                                      DWORD dwChannel,
                                      LPVOID lpvBuffer,
                                      DWORD dwBufferSize,
                                      LPDWORD lpdwBufferSizeFilled
                              );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The channel for which the data is read from the file

lpvBuffer The buffer which receives the data samples

dwBufferSize The maximum size of the buffer pointed to by 'lpvBuffer'

lpdwBufferSizeFilled A pointer to a DWORD receiving the actual number of bytes read from the file this number will be less or equal then *dwBufferSize*

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
BYTE byaDataBuffer[4096];
DWORD dwDataRead;

if (!DiiOpenBlockInputFile      (hDevice,
                                 0,
                                 "c:\\temp\\Channel0Dump.Bin"))
        return;         // unable to open file

if (DiiReadBlockInputFile       (hDevice,
                                 0,
                                 byaDataBuffer,
                                 4096,
                                 &dwDataRead))
        {       // succeeded, so something with the data        }

DiiCloseBlockInputFile          (hDevice,0);
```

Remarks

DiiCloseBlockInputFile

This function is called to close a previously opened block input file.

Declaration

```
BOOL DiiCloseBlockInputFile    (       HANDLE hDevice,
                                       DWORD dwChannel,
                               );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The channel for which the file is to be closed.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
BYTE byaDataBuffer[4096];
DWORD dwDataRead;

if (!DiiOpenBlockInputFile      (hDevice,
                                0,
                                “c:\\temp\\Channel0Dump.Bin”))
        return;         // unable to open file

if (DiiReadBlockInputFile       (hDevice,
                                0,
                                byaDataBuffer,
                                4096,
                                &dwDataRead))
        {       // succeeded, so something with the data        }

DiiCloseBlockInputFile          (hDevice,0);
```

Remarks

DiiPrepareBlockOutput

This function is used to prepare a single channel for block-based output.

Declaration

```
BOOL DiiPrepareBlockOutput   (      HANDLE hDevice,
                                    DWORD dwChannel,
                                    DWORD dwTriggerSource,
                                    DWORD dwTriggerValue
                             );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The index of the input channel to be scanned automatically (0 is the first channel)

dwTriggerSource Contains the trigger source for the acquisition.

Specify 0 – for a software emulated trigger,
Specify 1 – for a trigger based on an on-board timer
Specify 2 – for a trigger based on an external source, connected via a digital input channel

dwTriggerValue Contains either the frequency in acquisitions per second if an internal timer is used, or the channel number if an external signal is used for triggering an acquisition step.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
        // Forward declaration...
void ProcessDeviceNotification (LPDII_NOTIFICATION_DATA);


if (!DiiSetNotificationMethodCallBack (hDevice,
                                       &ProcessDeviceNotification
                                       ))
{
        return FALSE;
}

if (!DiiRequestNotification (hDevice, DII_DATA_BLOCK_SENT, 0))
{
        // could not request notification about blocks being written
        return FALSE;
}

if (!DiiPrepareBlockOutput (    hDevice,
```

```
                                        0,              // sample the first channel
                                        0,              // use internal timer as pacer
                                        30000           // try to send at 30 kHz
                        ))
{
        // Could not initialize the channel for block output.
        return FALSE;
}

if (!DiiPrepareBlockOutput (    hDevice,
                                        1,              // sample the second channel
                                        0,              // use internal timer as pacer
                                        10000           // try send at 10 kHz
                        ))
{
        // Could not initialize the channel for block output.
        return FALSE;
}
```

Remarks

This function is called to prepare a single channel for block output. It does not actually initiate any output, but it is used to setup the rate and the timing for block output.

DiiStartBlockOutput

This function is called to start the previously configured block output.

Declaration

```
BOOL DiiStartBlockOutput    (       HANDLE hDevice,
                                    );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
        // Forward declaration...
void ProcessDeviceNotification (LPDII_NOTIFICATION_DATA);


if (!DiiSetNotificationMethodCallBack (hDevice,
                                    &ProcessDeviceNotification
                                    ))
{
        return FALSE;
}

if (!DiiRequestNotification (hDevice, DII_DATA_BLOCK_SENT, 0))
{
        '' could not request notification about data blocks being written
        return FALSE;
}

if (!DiiPrepareBlockOutput (    hDevice,
                                0,                  // sample the first channel
                                0,                  // use internal timer as pacer
                                30000               // try to sample 30 kHz
                        ))
{
        '' Could not initialize the channel for block output.
        return FALSE;
}

if (!DiiPrepareBlockOutput (    hDevice,
                                1,                  // sample the second channel
                                0,                  // use internal timer as pacer
```

```
                                        10000                   // try to sample 10 kHz
                        ))
{
        '' Could not initialize the channel for block output.
        return FALSE;
}

if (!DiiStartBlockOutput  (         hDevice))
{
        '' cannot start block output
        return FALSE;
}
```

Remarks

This function should be called to start the background block output engine. It does not initiate any block output, but prepares the background threads and the associated hardware for block output.

DiiEndBlockOutput

This function is called to stop the block based output of data samples

Declaration

```
BOOL DiiEndBlockOutput      (       HANDLE hDevice,
                                    );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
        // Forward declaration...
void ProcessDeviceNotification (LPDII_NOTIFICATION_DATA);


if (!DiiSetNotificationMethodCallBack (hDevice,
                                       &ProcessDeviceNotification
                                       ))
{
        return FALSE;
}

if (!DiiRequestNotification (hDevice, DII_DATA_BLOCK_SENT, 0))
{
        '' could not request notification about data blocks being written
        return FALSE;
}

if (!DiiPrepareBlockInput (     hDevice,
                                0,              // sample the first channel
                                0,              // use internal timer as pacer
                                30000           // try to sample 30 kHz
                ))
{
        '' Could not initialize the channel for block input.
        return FALSE;
}

if (!DiiPrepareBlockOutput (    hDevice,
                                1,              // sample the second channel
                                0,              // use internal timer as pacer
                                10000           // try to sample 10 kHz
                ))
```

```
{
        ‘’ Could not initialize the channel for block output
        return FALSE;
}

if (!DiiStartBlockOutput  (        hDevice))
{
        ‘’ cannot start block output
        return FALSE;
}

BYTE byaDataBlock[4096];

DiiOutputBlock          (        hDevice,
                                 byaDataBlock,
                                 0,
                                 4096,
                                 TRUE); // contineously output this data block

        //
...     // do some work, i.e. processing the device notifications

if (!DiiEndBlockOutput (         hDevice))
{
        return FALSE;
}
```

Remarks

Use this function to stop a background data output in process

DiiOutputBlock

This function is called to actually output a data block

Declaration

```
BOOL DiiOutputBlock   (       HANDLE hDevice,
                              DWORD dwChannel,
                              LPVOID lpvBuffer,
                              DWORD dwBufferSize,
                              BOOL bContineous
                      );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The channel on which the block is to be output

lpvBuffer The buffer which is to be output

dwBufferSize The size of the buffer to be output

bContineous A flag that indicates whether the buffer shall continuously be output.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
        // Forward declaration...
    void ProcessDeviceNotification (LPDII_NOTIFICATION_DATA);


    if (!DiiSetNotificationMethodCallBack (hDevice,
                                           &ProcessDeviceNotification
                                           ))
    {
        return FALSE;
    }

    if (!DiiRequestNotification (hDevice, DII_DATA_BLOCK_SENT, 0))
    {
        '' could not request notification about data blocks being written
        return FALSE;
    }

    if (!DiiPrepareBlockInput (   hDevice,
                                  0,                    // sample the first channel
                                  0,                    // use internal timer as pacer
```

```
                                    30000                   // try to sample 30 kHz
                    ))
{
        '' Could not initialize the channel for block input.
        return FALSE;
}

if (!DiiPrepareBlockOutput (    hDevice,
                                    1,                      // sample the second channel
                                    0,                      // use internal timer as pacer
                                    10000                   // try to sample 10 kHz
                    ))
{
        '' Could not initialize the channel for block output
        return FALSE;
}

if (!DiiStartBlockOutput  (     hDevice))
{
        '' cannot start block output
        return FALSE;
}

BYTE byaDataBlock[4096];

DiiOutputBlock          (       hDevice,
                                    0,
                                    byaDataBlock,
                                    4096,
                                    TRUE); // contineously output this data block

        //
...     // do some work, i.e. processing the device notifications

if (!DiiEndBlockOutput (        hDevice))
{
        return FALSE;
}
```

Remarks

This function is used to actually output a data block through a channel. The block output interface has to be previously prepared and started, as shown in the example. The block output is actually performed in the background, such that the data buffer passed into the *DiiOutputBlock* function may not be accessed until the data block has been output.

The DII will send a DII_DATA_BLOCK_SENT notification to the application for every data block output. Through the lpvBuffer parameter of the LPDII_NOTIFICATION_DATA block, the user application can determine which data blocks have successfully been outputted.

DiiOutputBlockFromFile

This function is called to actually output a data block from a previously acquired data file

Declaration

```
BOOL DiiOutputBlock   (       HANDLE hDevice,
                              DWORD dwChannel,
                              LPCTSTR lpszFileName,
                      );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The channel on which the block is to be output

lpszFileName The filename containing the data samples to be output

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
        // Forward declaration...
    void ProcessDeviceNotification (LPDII_NOTIFICATION_DATA);


    if (!DiiSetNotificationMethodCallBack (hDevice,
                                           &ProcessDeviceNotification
                                           ))
    {
        return FALSE;
    }

    if (!DiiRequestNotification (hDevice, DII_DATA_BLOCK_SENT, 0))
    {
        '' could not request notification about data blocks being written
        return FALSE;
    }

    if (!DiiPrepareBlockInput (   hDevice,
                                  0,                    // sample the first channel
                                  0,                    // use internal timer as pacer
                                  30000                 // try to sample 30 kHz
                          ))
    {
        '' Could not initialize the channel for block input.
        return FALSE;
    }
```

```
if (!DiiPrepareBlockOutput (      hDevice,
                                  1,                    // sample the second channel
                                  0,                    // use internal timer as pacer
                                  10000                 // try to sample 10 kHz
                  ))
{
        ‘’ Could not initialize the channel for block output
        return FALSE;
}

if (!DiiStartBlockOutput  (       hDevice))
{
        ‘’ cannot start block output
        return FALSE;
}

DiiOutputBlockFromFile(           hDevice,
                                  0,
                                  “c:\\temp\\Channel0Dump.bin”,
                                  TRUE); // continuously output this data block

        //
...     // do some work, i.e. processing the device notifications

if (!DiiEndBlockOutput (          hDevice))
{
        return FALSE;
}
```

Remarks

This function is used to actually output a data block through a channel. The block output interface has to be previously prepared and started, as shown in the example. The data to be output is read from the file given. The file has to be previously created by the block input interface. (see *DiiPrepareBlockInput*).

The DII will send a DII_DATA_BLOCK_SENT notification to the application for every data block output.

DiiGetBlockOutputStatus

This function returns status information about the block output for a specific channel

Declaration

```
BOOL DIIEXPORT DiiGetBlockOutputStatus    (
                                          HANDLE hDevice,
                                          DWORD dwChannel,
                                          LPDWORD lpdwTotalSamples,
                                          LPDWORD lpdwSamplesPerSecond,
                                          LPDWORD lpdwBufferUnderruns
                                    );
```

Parameters

hDevice A valid device handle, previously obtained from DiiOpenDevice or DiiOpenNamedDevice

dwChannel The channel for which information is to be retrieved

lpdwTotalSamples
The total number of data samples (*not bytes!*) already sent

lpdwSamplesPerSecond
The average number of data samples per second.

lpdwBufferUnderruns
The number of buffer underruns encountered so far. Buffer underruns occur when the hardware is sending data at a faster rate than the DII can provide it. This is an error.

Return value

TRUE if successful, FALSE otherwise.

If an error occurred, GetLastError() may return the following values:

ERROR_INVALID_PARAMETER - The handle passed was invalid, or the channel specified is out of range.

Example

```
        //
        // see above example about how to setup & start block output

    DWORD dwTotalSamples;
    DWORD dwSamplesPerSecond;
    DWORD dwBufferUnderruns;

    if (DiiGetBlockOutputStatus (       hDevice,
                                        0,          // retrieve info for the first channel
                                        &dwTotalSamples,
                                        &dwSamplesPerSecond,
                                        &dwBufferUnderruns
                                ))
    {
        // evaluate the statistical information
    }
```

Remarks

This function returns statistical information during a running block output. The user application can observe that all possible data has been correctly sent by evaluating the buffer underruns error. A buffer underrun occurs if the hardware sends data at a faster rate than the DII can provide the data to the hardware.

9. The Dynamic Industrial Interface OCX/ActiveX Control

The DII-OCX is shipped with the Dynamic Industrial Interface Drivers and is registered properly for use by many applications, such as Visual Basic, or Delphi. In many cases, application programming utilizing the DII-OCX is more quickly and more efficient, since many technical issues, such as parameter conversion is kept away from the programmer. The OCX offers the same functionality as the DII-DLL. You may still use the DLL in your C/C++ applications, for greater convenience.

Through OLE-Automation, the OCX exposes properties and methods, which can easily be changed or invoked from high-level languages. The following pages include a detailed description of each method and property offered by the OCX.

9.1. Properties

The following properties are exposed by the OCX control:

Property

DeviceName

Type

String

Persistent

Parameters

None

Example (Visual Basic)

```
‘’ The Example assumes you have a form containing the DII-OCX control named “DiiDevice”.
‘’ See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = “Device0”

If Not DiiDevice.DeviceOpened Then

        MsgBox “Unable To Open Device named Device0”

End If
```

Remarks

This property contains the device name to be opened by an instance of the OCX. As soon as this property is being changed, the OCX closes the existing devices and attempts to reopen the device with the new name specified.

You may also use the OCX method "SelectDevice" to display a dialog box in which the user can select a device name to operate upon.

In order to determine if the device name actually matched a device installed in your machine and is ready for access, you may use the property "DeviceOpened", which is TRUE, if the device was opened successfully, or FALSE otherwise.

Property

Exclusive

Type

Boolean

Persistent

Parameters

None

Example (Visual Basic)

```
‘’ The Example assumes you have a form containing the DII-OCX control named “DiiDevice”.
‘’ See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = “Device0”

DiiDevice.Exclusive = True

If Not DiiDevice.DeviceOpened Then

    MsgBox “Unable To Open Device named Device0”

End If
```

Remarks

This property determines how to open an adapter, either exclusively or non-exclusively. Setting this property to TRUE will prevent other applications or other OCXs from opening the device with the same name.

Property

DeviceOpened

Type

Boolean

Read-Only

Parameters

None

Example (Visual Basic)

```
‘’ The Example assumes you have a form containing the DII-OCX control named “DiiDevice”.
‘’ See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = “Device0”

If Not DiiDevice.DeviceOpened Then

        MsgBox “Unable To Open Device named Device0”

End If
```

Remarks

This property is set to TRUE, whenever the OCX has successfully opened a device, FALSE otherwise.

Property

DigitalChannels

Type

long

Read-Only

Parameters

None

Example (Visual Basic)

```
‘’ The Example assumes you have a form containing the DII-OCX control named “DiiDevice”.
‘’ See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = “Device0”

If Not DiiDevice.DeviceOpened Then

        MsgBox “Unable To Open Device named Device0”

Else

        If DiiDevice.DigitalChannels = 0 Then

                MsgBox “This device cannot do digital I/O”

        End If

End If
```

Remarks

This property contains the number of digital channels for the currently opened Industrial I/O device, as specified in the property "DeviceName".

Note: You can also use the *DigitalInputChannels* and *DigitalOutputChannels* properties to differentiate between input and output channels.

Property

DigitalInputChannels

Type

long

Read-Only

Parameters

None

Example (Visual Basic)

```
‘’ The Example assumes you have a form containing the DII-OCX control named “DiiDevice”.
‘’ See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = “Device0”

If Not DiiDevice.DeviceOpened Then

        MsgBox “Unable To Open Device named Device0”

Else

        If DiiDevice.DigitalInputChannels = 0 Then

                MsgBox “This device does not have digital input lines”

        End If

End If
```

Remarks

This property contains the number of digital input channels for the currently opened Industrial I/O device, as specified in the property "DeviceName".

Property

DigitalOutputChannels

Type

long

Read-Only

Parameters

None

Example (Visual Basic)

```
‘’ The Example assumes you have a form containing the DII-OCX control named “DiiDevice”.
‘’ See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = “Device0”

If Not DiiDevice.DeviceOpened Then

        MsgBox “Unable To Open Device named Device0”

Else

        If DiiDevice.DigitalOutputChannels = 0 Then

                MsgBox “This device does not have digital output lines”

        End If

End If
```

Remarks

This property contains the number of digital output channels for the currently opened Industrial I/O device, as specified in the property "DeviceName".

Property

AnalogChannels

Type

long

Read-Only

Parameters

None

Example (Visual Basic)

```
‘’ The Example assumes you have a form containing the DII-OCX control named “DiiDevice”.
‘’ See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = “Device0”

If Not DiiDevice.DeviceOpened Then

        MsgBox “Unable To Open Device named Device0”

Else

        If DiiDevice.AnalogChannels = 0 Then

                MsgBox “This device cannot do analog I/O”

        End If

End If
```

Remarks

This property contains the number of analog channels for the currently opened Industrial I/O device, as specified in the property "DeviceName".

Note: You can also use the *AnalogInputChannels* and *AnalogOutputChannels* properties to differentiate between input and output channels.

Property

AnalogInputChannels

Type

long

Read-Only

Parameters

None

Example (Visual Basic)

```
‘’ The Example assumes you have a form containing the DII-OCX control named “DiiDevice”.
‘’ See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = “Device0”

If Not DiiDevice.DeviceOpened Then

        MsgBox “Unable To Open Device named Device0”

Else

        If DiiDevice.AnalogInputChannels = 0 Then

                MsgBox “This device does not have analog input channels”

        End If

End If
```

Remarks

This property contains the number of analog input channels for the currently opened Industrial I/O device, as specified in the property "DeviceName".

Property

AnalogOutputChannels

Type

long

Read-Only

Parameters

None

Example (Visual Basic)

```
‘’ The Example assumes you have a form containing the DII-OCX control named “DiiDevice”.
‘’ See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = “Device0”

If Not DiiDevice.DeviceOpened Then

        MsgBox “Unable To Open Device named Device0”

Else

        If DiiDevice.AnalogOutputChannels = 0 Then

                MsgBox “This device does not have analog output channels”

        End If

End If
```

Remarks

This property contains the number of analog output channels for the currently opened Industrial I/O device, as specified in the property "DeviceName".

Property

Resolution

Type

long

Read-Only

Parameters

None

Example (Visual Basic)

```
‘’ The Example assumes you have a form containing the DII-OCX control named “DiiDevice”.
‘’ See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = “Device0”

If Not DiiDevice.DeviceOpened Then

    MsgBox “Unable To Open Device named Device0”

Else

    MaximumResolution = DiiDevice.Resolution

End If
```

Remarks

This property contains the resolution in bits for the currently opened Industrial I/O device, as specified in the property "DeviceName".

Property

ChannelGain

Type

double

Parameters

IsInputChannel (Boolean)

ChannelIndex (Long)

Example (Visual Basic)

```
'' The Example assumes you have a form containing the DII-OCX control named "DiiDevice".
'' See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = "Device0"

If Not DiiDevice.DeviceOpened Then

        MsgBox "Unable To Open Device named Device0"

Else

        Dim CurrentGain As Double
        CurrentGain = DiiDevice.ChannelGain (True,0)
                '' Reads the channel gain from the first channel

        DiiDevice.ChannelGain(True,0) = 10.0
                '' Sets the amplification for the first channel to 10.

End If
```

Remarks

This property sets or returns the channel gain/amplification that is configured for a specific analog channel. For further information consult the description of the function DiiSetChannelGain and DiiGetChannelGain.

Property

ChannelBipolar

Type

double

Parameters

IsInputChannel (Boolean)

ChannelIndex (Long)

Example (Visual Basic)

```
'' The Example assumes you have a form containing the DII-OCX control named "DiiDevice".
'' See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = "Device0"

If Not DiiDevice.DeviceOpened Then

        MsgBox "Unable To Open Device named Device0"

Else

        Dim IsChannelBipolar As Boolean
        IsChannelBipolar = DiiDevice.ChannelBipolar (True,0)
                '' Reads the bipolar mode from the first channel

End If
```

Remarks

This property sets or returns the bipolar mode that is configured for a specific analog channel. For further information consult the description of the function DiiSetChannelBipolar and DiiGetChannelBipolar.

Property

RealAnalogChannel

Type

double

Parameters

ChannelIndex (Long)

Example (Visual Basic)

```
‘’ The Example assumes you have a form containing the DII-OCX control named “DiiDevice”.
‘’ See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = “Device0”

If Not DiiDevice.DeviceOpened Then

        MsgBox “Unable To Open Device named Device0”

Else

        Dim CurrentValue As Double

        CurrentValue = DiiDevice.RealAnalogChannel(0)
                ‘’ Reads the current value of the first analog channel

        DiiDevice.RealAnalogChannel(1) = 3.5
                ‘’ Sets the second analog output channel to 3.5 Volts.

End If
```

Remarks

This property can be used to easily read and write real analog values to analog input and output channels. Note that if the function fail, an exception will be thrown. For further information, consult the appropriate Dii functions DiiSetRealAnalogChannel and DiiGetRealAnalogChannel.

Property

NotificationMask

Type

long

Parameters

None

Example (Visual Basic)

```
'' The Example assumes you have a form containing the DII-OCX control named "DiiDevice".
'' See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = "Device0"

If Not DiiDevice.DeviceOpened Then

    MsgBox "Unable To Open Device named Device0"

Else

    DiiDevice.NotificationMask = &H2
                '' Enables Signal Change Notifications

                '' Check, if notifications were activated:
    If Not DiiDevice.NotificationsActive Then
        MsgBox "This device does not support notifications"
    End If

                '' Disable notifications again.
    DiiDevice.NotificationMask = 0

End If
```

Remarks

Setting this property to a non-zero value, will enable the requested notifications. This is equivalent to calling the "RequestNotification". If NotificationMask is assigned to zero, notifications are disabled. This is equivalent to calling "CancelNotification"

Property

NotificationsActive

Type

long

Read-Only

Parameters

None

Example (Visual Basic)

```
'' The Example assumes you have a form containing the DII-OCX control named "DiiDevice".
'' See the following Chapter for information about how to use the OCX from Visual Basic

DiiDevice.DeviceName = "Device0"

If Not DiiDevice.DeviceOpened Then

    MsgBox "Unable To Open Device named Device0"

Else

    DiiDevice.NotificationMask = &H2
                '' Enables Signal Change Notifications

                '' Check, if notifications were activated:
    If Not DiiDevice.NotificationsActive Then
        MsgBox "This device does not support notifications"
    End If

                '' Disable notifications again.
    DiiDevice.NotificationMask = 0

End If
```

Remarks

This property indicates whether notifications were successfully enabled or disabled. If set to True, the OCX can fire "Notify" events at any time.
If set to False, the OCX does not fire "Notify" events.

9.2. Methods

The methods exposed by the OCX control basically correspond to the respective DII functions of the Dynamic Industrial Interface already covered in the previous chapter. For the following methods, please consult the appropriate description in the function overview in previous chapter.

SetDigitalBit	->	DiiSetDigitalBit
GetDigitalBit	->	DiiGetDigitalBit
SetDigitalByte	->	DiiSetDigitalByte
GetDigitalByte	->	DiiGetDigitalByte
Set8255Config	->	DiiSet8255Config
SetAnalogChannel	->	DiiSetAnalogChannel
GetAnalogChannel	->	DiiGetAnalogChannel
SetRealAnalogChannel	->	DiiSetRealAnalogChannel
GetRealAnalogChannel	->	DiiGetRealAnalogChannel
SetChannelRange	->	DiiSetChannelRange
GetChannelRange	->	DiiGetChannelRange
GetDeviceChannelRange	->	DiiGetDeviceChannelRange
SetTimerConfig	->	DiiSetTimerConfig
LoadTimer	->	DiiLoadTimer
PrepareBlockInput	->	DiiPrepareBlockInput
StartBlockInput	->	DiiStartBlockInput
EndBlockInput	->	DiiEndBlockInput

(etc)

For specific examples on how to utilize these functions, consult the sample applications supplied.

9.2. Events

In the current version, the DII-OCX supports sending events for interrupt driven cards. These events correspond to notifications from the DII-DLL. Notifications can be send whenever a signal changes on the card, or if a counter drops to zero.

The DII-OCX exposes one central event “Notify”. When a Notify event is fired, the following parameters are available:

```
Notify(ByVal NotificationMask As Long, _
       ByVal NotificationParam As Long, _
       ByVal DataBuffer As Variant, _
       ByVal BufferSize As Long)
```

Parameters

NotificationMask
A bitmask containing the notifications sent. In this mask, one or more bits corresponding to the previously setup NotificationMask property.

NotificationParam
Contains parameters for the notification. The interpretation of this parameter depends on the notification fired. See the description of the DII_NOTIFICATION_DATA for further explanation.

DataBuffer A variant containing a binary buffer of data samples accompanying the event. This is used for background sampling of data.

BufferSize The size of the buffer in bytes.

Example (Visual Basic)

```
‘’ The Example assumes you have a form containing the DII-OCX control named “DiiDevice”.
‘’ See the following Chapter for information about how to use the OCX from Visual Basic


Private Sub Form_Load()

    DiiDevice.DeviceName = “Device0”

    If Not DiiDevice.DeviceOpened Then
        MsgBox “Unable To Open Device named Device0”
    Else
        DiiDevice.NotificationMask = &H2
                ‘’ Enables Signal Change Notifications

                ‘’ Check, if notifications were activated:
        If Not DiiDevice.NotificationsActive Then
            MsgBox “This device does not support notifications”
        End If
    End If
End Sub

Private Sub Form_Unload()

        ‘’ Disable notifications again.
```

```
        DiiDevice.NotificationMask = 0
End Sub

Private Sub DiiDevice_Notify(ByVal NotificationMask As Long, _
                  ByVal NotificationParam As Long, _
                  ByVal DataBuffer As Variant, _
                  ByVal BufferSize As Long)
                "
                " This function is called whenever there is a notification
                " sent by the device.

                " Check the type of notification:

        If ((NotificationMask And &H2) <> 0) Then

                If ((NotificationParam And &H1) <> 0) Then
                        '' Signal 1 changed.
                End If

                If ((NotificationParam And &H2) <> 0) Then
                        '' Signal 2 changed.
                End If

                If ((NotificationParam And &H4) <> 0) Then
                        '' Signal 3 changed.
                End If

                If ((NotificationParam And &H8) <> 0) Then
                        '' Signal 4 changed.
                End If

                        '' ...
        End If

        If ((NotificationMask And &H20) <> 0) Then
                "
                " It is a down-counter reached zero notification
                Beep
        End If
End Sub
```

10. Using the Dynamic Industrial Interface with different programming languages

This chapter provides an overview about how to best utilize the Dynamic Industrial Interface in various programming languages.

If you experience difficulties calling the Dynamic Industrial Interface functions from your programming language, or are using a programming language not covered in this documentation, please feel free to visit our web-site, to which we will post updated information regarding DII programming issues. You may also contact our technical support through our web-site.

10.1. C++

Since the Dynamic Industrial Interface DLL was also developed using C++, you may easily use it to access Industrial I/O devices.
For this purpose, a C++ header file ("Dii.h") as well as an import library ("Dii.lib") are being shipped with the interface library. Make sure that you have installed the development release, not the retail release, which does not include support programming files.

In your C/C++ source code files, just include the "Dii.h" include file, then you can use any of the functions provided by the Dynamic Industrial Interface DLL. Be sure to include the import library "Dii.lib" during the linking step of your application, so your applications successfully references the actual interface DLL.

10.2. Visual Basic

Note: Since the Dynamic Industrial Interface is fully 32-bit compliant, only 32-bit versions of Visual Basic are supported. Specifically, Version 4.0 and Version 5.0 are tested and supported.

If you are using Visual Basic to access any I/O Devices supported by the Dynamic Industrial Interface, you basically have two choices:
You can either call the Dii DLL directly, or use the OCX. We recommend you use the OCX for easier access of the data acquisition functions supported.
In order to use the OCX control, be sure you have successfully installed the Dynamic Industrial Interface on your computer. Then add the OCX control to your Visual Basic Toolbar, using the "Projects/ Component..." menu item (Visual Basic V5.0).

To access a specific device, you can just drag the DII OCX control from the Visual Basic Toolbar onto your form, and select the device name in the properties section. All further programming of DII supported cards can be done exactly like programming any other OCX. Refer to the listing of OCX properties and methods in Chapter 9 for more specific information about available functions.

You may also consult the Visual Basic sample application supplied in the “DiiDemoVB” subdirectory of your installation directory for more information about using Visual Basic to access the Dynamic Industrial Interface.

10.3. Borland Delphi

Note: Since the Dynamic Industrial Interface is fully 32-bit compliant, only 32-bit versions of Delphi are supported. Specifically Version 2.0 is tested and supported.

The Dynamic Industrial Interface can easily be used using the Delphi programming language by using the enclosed OCX/ActiveX control. You will first have to add the OCX control to your project by selecting “Component/Install...”, then choose to install an “OCX” component. Delphi will create a unit (.pas) for you which exports a class/type named “TDiiOcxCtrl”. This new type encapsulates the DII OCX, and you can refer to any properties of the OCX easily.

You can then drag the OCX control onto your form and graphically edit all properties.

You may also consult the Delphi sample application supplied in the “DiiDemoDelphi” subdirectory of your installation directory for more information about using Delphi to access the Dynamic Industrial Interface.

11. Technical Support And Feedback

We believe that customer input is the most valuable source for creating successful products.

We continuously update and extend the Dynamic Industrial Interface with new functionality, for specific devices, for specific applications, to meet your specific needs, and provide supportive products around the DII.