Table of Contents

Getting Started

The Serial Port Monitoring Control library provides the serial port monitoring functionality for your code. The library lets you enumerate all installed serial devices, including, but not limited to serial ports and modems. All kind of PnP serial devices as well as virtual devices are also supported. After you get an instance of the serial device, you can create a monitor object and attach it to the device to receive monitored data and events. The Monitor object can be attached to the device at any time, no matter if the device is being currently used or not. You can also detach from the device at any time. Below you will find the short list of terms used throughout this documentation.

SPMC, The Library, library, control

The Serial Port Monitoring Control library

User or client code

Code in any language that instantiates the SPMC and calls its methods

The monitored application

The application that has the monitored serial device open.

Library Features

The HHD Software Serial Port Monitoring Control library offers the following functionality:

- Support for PnP and virtual serial devices with hot-plug and hot-unplug functionality.
- Ability to work with any software that opens a serial port and initiates communication through it.
- Interception of all data read from and written to the serial device.
- Interception and detailed decoding of all serial input/output control codes (IOCTLs).
- Full compatibility with ACPI features.
- Support for Windows 7, Windows 8, Windows 8.1 and Windows 10 (32-bit and 64-bit) as well as corresponding server versions.
- Provides two high-performance mechanisms for native code and one high-compatibility interface for managed code.
- High compatibility with all modern development environments, including Microsoft Visual Studio and Embarcadero RAD Studio.
- Library client code can be written in any language, including C++, Delphi, C#, VB.NET and any other CLR-compatible language.

Licensing

This section contains the list of licenses the Serial Port Monitoring Control is distributed under. Please note that the licensing is subject to change. For a most recent information please visit the HHD Software Web site.

You may find the detailed information on Serial Port Monitoring Control licensing on-line. Click on a link below to read the corresponding license agreement.

License type	Single developer	Unlimited number of developers	Distribution rights included
Single Developer & Distribution	X		X
Site Developer & Distribution		X	X

Installation

The library is distributed as a single executable file which is signed by the HHD Software, Ltd. You can use the operating system's provided tools to verify the digital signature to make sure the file is delivered

unmodified by any third-party and is free of transmission errors.

The library installer, when launched, asks you to provide the library installation path, or to accept the default. You can also choose whether you need the library samples to be installed or not.

After installation, the following structure appears in destination folder:

`bin/x64`

- `hhdspmc.dll` (64-bit version of ActiveX control DLL)
- `DIFxAPI64.dll` (64-bit driver installation framework redistributable component)

`bin/x86`

- `hhdspmc.dll` (32-bit version of ActiveX control DLL)
- `DIFxAPI32.dll` (32-bit driver installation framework redistributable component)

`doc`

- `hhdspmc.chm` (this documentation)

`drivers`

- `hhdspmc.inf` (Driver installation information file)
- `hhdspmc_x86.cat` (32-bit driver catalog file)
- `hhdspmc_x64.cat` (64-bit driver catalog file)
- `hhdspmc32.sys` (32-bit serial filter driver)
- `hhdspmc64.sys` (64-bit serial filter driver)

`inc`

- `hhdspmc.h` (Compiled IDL file)
- `hhdspmc.idl` (IDL file with library classes and interfaces)

`lib/x64`

- `hhdspmc.lib` (64-bit import library)

`lib/x86`

- `hhdspmc.lib` (32-bit import library)

`redist/Manual`

- `spmc_redist.exe` (SPMC redistributable package)

`redist/Merge Module/x64`

- `spmc_msm_x64.msm` (64-bit Windows Installer Merge Module)

`redist/Merge Module/x86`

- `spmc_msm_x86.msm` (32-bit Windows Installer Merge Module)

`Samples`

Contains all SPMC sample projects and solutions

Several files (dynamic-link libraries, drivers and import libraries) are available as 32-bit and 64-bit. When SPMC is installed on 64-bit machine, both 64-bit and 32-bit libraries are registered, allowing all kind of user code to access the library.

On 32-bit machine, on the other hand, only 32-bit library is registered, and you can only run 32-bit client code.

Library Redistribution Policy

Library redistribution policy largely depends on the purchased license. Detailed information about library redistribution can be found in the Licensing topic. General information is also provided below.

General Library Distribution Information

The SPMC library can be distributed in two ways:

Distribution of the original SPMC installation package.

You are allowed to distribute the original library installation package provided the following is true: the `hhdspmc.exe` file is not modified by any means and is accompanied by a link to HHD Software web site. This can be an Internet URL file, a readme file in the same folder or same archive where the `hhdspmc.exe` file is located on distribution media. If the file is distributed on the Web Site, the link to HHD Software web site must be on the same page as the link to the `hhdspmc.exe` file.

The `hhdspmc.exe` file, as well as all other files available on HHD Software Web Site, is digitally signed. Please verify that the digital signature is still valid before distributing the file. If, on the other side, you somehow received the Serial Port Monitoring Control library and found that signature validation algorithm fails, please be so kind to notify us via our contact page.

Library Redistribution

Two ways of redistributing the library are provided by the current version: manual redistribution and Windows Installer merge module.

Manual Redistribution

To redistribute the library with your application, use the `spmc_redist.exe` file, located in the `redist\Manual` folder.

This is a Win32 executable without any external references. It supports all operating systems starting from Windows 7 (both 32-bit and 64-bit).

Launch this executable on a target computer to install or uninstall the SPMC library and accompanying filter driver. It automatically installs 32-bit or 64-bit version of components. Below is executable's command line parameters:

```
Command Prompt
spmc_redist.exe [/q] [/u] [/t <path>] [/?]
```

Parameter	Description
`/q`	Quiet mode. Do not display any user interface. Note that on some operating systems, driver installation prompt may still be displayed.
`/u`	Uninstall the library. If not specified, performs library installation.
`/t <path>`	Use the given installation path instead of the default one.
`/?`	Display supported command line parameters.

`spmc_redist.exe` must be launched by a user with administrative privileges. On Windows 7 and later operating systems, the user must be elevated. Library installation may be performed by more restricted user if she has write access to destination folder, has write access to system drivers folder (usually `\Windows\System32\Drivers`) and has been assigned an `Install Device Drivers` privilege.

If `/q` switch is not specified, a message box with a short description of the result of operation is displayed.

Upon exit, `spmc_redist.exe` returns an error code, which must be interpreted as described below:

Error code	Meaning
`0 (S_OK)`	Library installation/uninstallation has been successful.
`1 (S_FALSE)`	Library installation/uninstallation has been successful. Reboot is required to complete the installation.
any other	Any other code must be treated as a standard `HRESULT` value.

Windows Installer Merge Module

If your application is packaged with Windows Installer, you may use the merge modules provided in the `redist\Merge Module` folder. Use the correct module for your target platform (32-bit or 64-bit).

Activating the Redistributed Library

Library activation is performed in your installation code by calling one of the ISerialMonitor.InstallLicense, ISerialMonitor.InstallLicenseInMemory or ISerialMonitor.InstallLicenseInMemoryNative methods.

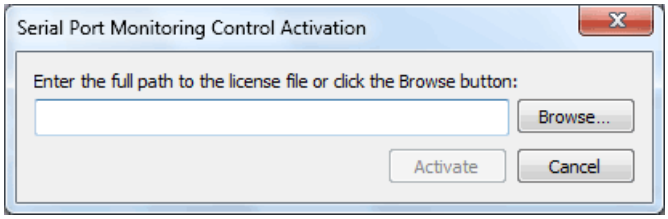
You are explicitly prohibited from charging any fee for all kinds of library distribution!

Activating Serial Port Monitoring Control

This section provides a complete description of the steps you need to follow in order to activate your copy of the HHD Software Serial Port Monitoring Control.

First, you need to have an account in our internal database, which means that the product must already be purchased. If you haven't purchased it yet, please follow to the product's homepage.

Below is the Activation dialog.



You need to provide a full path to the license file in order to activate the SPMC.

You can enter the path manually, or click the **Browse** button.

Using SPMC

Usage Environments

This documentation defines the usage environment as a programming language or development platform, for which it is possible to use the Serial Port Monitoring Control library. Theoretically, there are unlimited languages and platforms that can use the library, because it supports both pure `COM` and `OLE` automation interfaces, which makes it highly portable. For clarity, in this documentation we focus on the following two environments:

Native Environment

Native environment means the language that is binary compatible with `COM`, more precisely, the one that supports `COM` local interfaces. You can create a native client in `C/C++` languages, `Borland Delphi` and others. As usual, if you are looking for the highest performance and lowest overhead, while keeping your executable as small as possible, the native environment is the right choice for you.

The native client must load the library as an in-proc component and make sure to use the free threaded marshaller. In fact, the so-called native interfaces in the library require that your runtime does not provide the marshaller for the interface - they are intended to be called directly. INativeListener is the only pure native interface in the library. All other interfaces are dual and can be called through any marshaller (or without one).

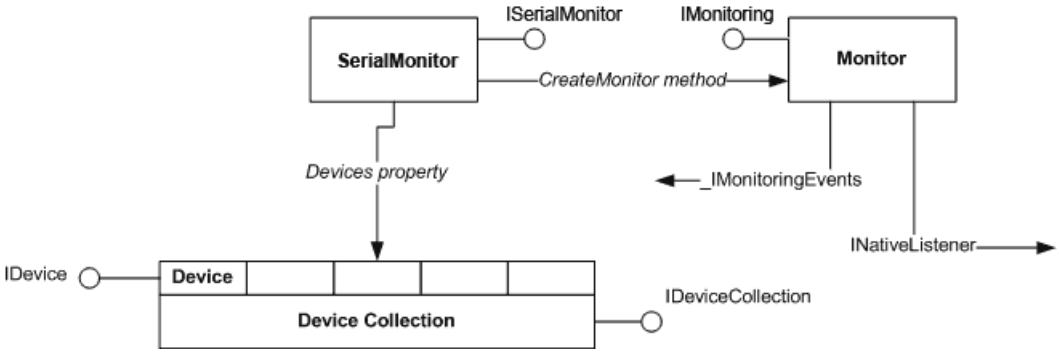
Managed Environment

Under managed environment we understand all scripting and similar languages. They include `JavaScript`, `VB Script`, `Visual Basic`, `Java` and the whole .NET platform. All these languages and tools are capable of working with `OLE` automation-compatible interfaces. The SPMC library supports managed clients, providing the highest possible performance while keeping the library usage as simple as possible.

Please note, that you can also create the client in "native" languages, such as `C++` or `Delphi`, that in fact uses the managed version of the library interfaces. While this is possible and actually may simplify your code and boost your development time, the performance will be slightly less, compared to the one in pure native environment.

General Guidelines

The library has the following structure:



Native Conventions

Dual Interfaces

ISerialMonitor, IDeviceCollection, IDevice and IMonitoring interfaces are declared *dual*. To use their binary-compatible (local) part in native code, include the `hhdspmc.h` file in your project. Remember, that the library must be loaded into your process address space to use the local interfaces without a marshaller.

INativeListener Local Interface

The INativeListener interface is a local interface, so it can only be used in the native code. Unlike other interfaces declared in the SPMC, the library does not have any object that implements this interface. Instead, you must implement this interface in your native listener object. Each **Monitor** object supports adding one or more native listeners via the IMonitoring.AddNativeListener method call.

The monitor object is capable of providing two different protocols for each native listener. Your implementation must support exactly one of these protocols. If you support the first (raw) protocol, you will be able to achieve the highest possible speed, but will have to parse the monitored data in your own code. If you choose to support the second (standard) protocol, the library will parse each monitored request for you and call the appropriate method in the `INativeListener` interface for each request type.

Time Values

Every standard protocol method in the `INativeListener` interface accepts the `FILETIME *` value as a first parameter. This parameter points to the library-allocated `FILETIME` structure containing the time of the monitored event. Time is stored in UTC. Do not modify the contents of the variable, treat it as read only.

INativeListener Methods Parameters

Each standard protocol method in the `INativeListener` interface provides one or more parameters which contain or point to the parsed request data. The first parameter is always a pointer to a `FILETIME` structure, containing the UTC time of the event, while others (if present) contain the parsed request data. Consult the documentation for individual methods to get more information.

The general rule states that all data passed to the `INativeListener` methods should be considered read only.

Managed Conventions

Dual Interfaces

All managed environments are capable of using dual interfaces. Unlike the native environments, which use the local, binary-compatible part of the dual interface, the managed environment uses the `IDispatch` interface to call methods and query/set properties. ISerialMonitor, IDeviceCollection, IDevice and IMonitoring interfaces are all dual interfaces and can be easily used in managed environments.

Event Interfaces

The Serial Port Monitoring Control library has two event interfaces:_ISerialMonitorEvents and _IMonitoringEvents. Different managed environments support different styles of connecting event handlers to event sources. Please consult your language documentation to find out how to handle OLE automation events. Native clients are also able to attach event handlers to event sources. Please consult the MSDN for more information, or look at the included sample code for an example.

How To

How To Initialize the SPMC Library

This section describes the steps you need to carry in order to successfully initialize the SPMC library.

Native Environment

1. Use the `#import` directive to bring definitions into the scope:

```
C++
#import "full_path_to_hhdspmc.dll"
```

or

```
C++
#import "progid:hhdspmc.SerialMonitor.1.2"
```

or

```
C++
#import "libid:DD962786-3734-4BE3-B375-5E6F3FD37E37" version("1.2")
```

2. Declare the pointer to the `ISerialMonitor` interface.

```
C++
ISerialMonitorPtr pSerialMonitor;
```

3. Create the instance of the `SerialMonitor object`.

```
C++
pSerialMonitor.CoCreateInstance(__uuidof(SerialMonitor));
```

Remember that you cannot create more than one `SerialMonitor` object in single process. Although, you can have as many Monitor objects as you need.

Managed Environment

Use your language-provided tools to add the reference to the Serial Port Monitoring Control library into your project.

How To Enumerate Serial Devices

This section describes the steps you need to carry in order to enumerate the serial devices installed on the computer.

1. Initialize the SerialMonitor object, as described in the this tutorial.
2. Obtain the pointer to the IDeviceCollection interface of the serial device collection object by taking the value of the Devices property:

```
C++
IDeviceCollectionPtr pDeviceCollection = pSerialMonitor->Devices;
```

```
C#
DeviceCollection devices = sm.Devices;
```

3. Get the value of the Count property:

```
C++
long Count = pDeviceCollection->Count;
```

```
C#
uint Count = devices.Count;
```

4. Cycle through all items of the collection:

```
C++
for (long i = 0; i < Count; ++i)
{
    IDevicePtr pDevice = pDeviceCollection->Item[CComVariant {i}];
    // ...
}
```

```
C#
foreach (var device in devices)
{
    // ...
}
```

How To Retrieve the Serial Device Properties

This section describes the steps you need to carry in order to retrieve the properties of the serial device.

1. Obtain the IDevice pointer for the device in question.
2. Take the values of the Name, ConnectionString, Icon, Port, Present and Type properties.

```
C++
_bstr_t Name, ConnectionString, Port;
VARIANT_BOOL Present;
DeviceType Type;
HICON hIcon;

Name = pDevice->Name;
ConnectionString = pDevice->ConnectionString;
Port = pDevice->Port;
Present = pDevice->Present;
Type = pDevice->Type;
hIcon = pDevice->Icon;
```

```
C#
string Name = device.Name;
string ConnectionString = device.ConnectionString;
string Port = device.Port;
bool Present = device.Present;
DeviceType type = device.Type;
```

How To Create a Monitor Object

This section describes the steps you need to carry in order to create a monitor object.

1. Initialize the `SerialMonitor` object, as described in the this tutorial.
2. Call the CreateMonitor method to create a monitor object and receive the IMonitoring interface.

```
C++
IMonitoringPtr pMonitor = pSerialMonitor->CreateMonitor();
```

```
C#
Monitor monitor = sm.CreateMonitor();
```

You can create as many monitor objects as you need. Each Monitor object can be attached to one serial device at a time and can have as many native listeners or event handlers attached, as you need.

How To Receive Monitored Events

This section describes the steps you need to carry in order to receive monitored events in your code.

1. Obtain the `Monitor` object, as described in this tutorial.

2. Add your listener object (for native code) to the `Monitor` object or connect event handlers (for managed code):

C++
```
class CMyListener : CComObjectRoot<CMyListener>, public INativeListener
{
public:
    // override all pure virtual methods in the INativeListener
};

...

CComObject<CMyListener> pMyListenerObject;
CComObject<CMyListener>::CreateInstance(&pMyListenerObject);
CComPtr<INativeListener> pMyListener;
pMyListenerObject->QueryInterface(&pMyListener);
pMonitor->AddNativeListener(pMyListener);
```

C#
```
monitor.OnOpen += new hhdspmcLib._IMonitoringEvents_OnOpenEventHandler(monitor_OnOpen);
monitor.OnPurge += new hhdspmcLib._IMonitoringEvents_OnPurgeEventHandler(monitor_OnPurge)
monitor.OnRead += new hhdspmcLib._IMonitoringEvents_OnReadEventHandler(monitor_OnRead);
...
```

3. Attach the Monitor object to the serial device:

   - Attach to the given serial device:

C++
```
pMonitor->Connect(CComVariant(pDevice));
```

C#
```
monitor.Connect(device);
```

   - Attach to the serial device connected to the COM port:

C++
```
pMonitor->Connect(CComVariant(L"COM1"));
```

C#
```
monitor.Connect("COM1");
```

   - Attach to the next serial device the user plugs into the computer:

C++
```
pMonitor->Connect(CComVariant {VT_EMPTY});
```

C#
```
monitor.Connect();
```

Reference

ISerialMonitor Interface

Description

This is the first interface you get from the hhdspmc library. It is used to get the list of installed serial devices and create `Monitor` objects.

Declaration

TypeScript

```typescript
interface ISerialMonitor {
    // Properties
    // Methods
    CreateMonitor(): IMonitoring;
    InstallLicense(LicenseFile: string): void;
}
```

C#

```csharp
public interface ISerialMonitor
{
    // Properties
    IDeviceCollection Devices { get; set; }
    // Methods
    IMonitoring CreateMonitor();
    void InstallLicense(string LicenseFile);
    void DisplayActivationDialog(ulong WindowHandle);
    void InstallLicenseInMemory(Array Data);
}
```

C++

```cpp
struct ISerialMonitor : IDispatch
{
    // Properties
    IDeviceCollectionPtr Devices;  // get set
    // Methods
    IMonitoringPtr CreateMonitor();
    HRESULT InstallLicense(_bstr_t LicenseFile);
    HRESULT DisplayActivationDialog(HWND WindowHandle);
    HRESULT InstallLicenseInMemory(SAFEARRAY(byte) Data);
    HRESULT InstallLicenseInMemoryNative(const UCHAR * pData, ULONG Size);
};
```

Examples

This example shows you how to create a SerialMonitor object:

TypeScript

```typescript
let serialMonitor = new ActiveXObject()
```

C#

```csharp
var pSerialMonitor = new hhdspmcLib.SerialMonitor();
```

```
C++
#import "hhdspmc.dll"

// ...

hhdspmcLib::ISerialMonitorPtr pSerialMonitor;
if (SUCCEEDED(pSerialMonitor.CreateInstance(__uuidof(SerialMonitor))))
{
    // ... work with object
}
```

ISerialMonitor Properties

Devices

```
TypeScript
// This property is not available in scripting environment
```

```
C#
IDeviceCollection Devices { get; set; }
```

```
C++
IDeviceCollectionPtr Devices;  // get set
```

Description

Use this property to get the serial devices collection. Each installed serial device is present in this collection. If the serial device is currently not plugged to the computer, it still exists in this collection, but its Present property equals to `false`.

ISerialMonitor Methods

CreateMonitor

```
TypeScript
CreateMonitor(): IMonitoring;
```

```
C#
IMonitoring CreateMonitor();
```

```
C++
IMonitoringPtr CreateMonitor();
```

Description

Creates a new monitor object that is ready to be attached to serial device for monitoring.

InstallLicense

```
TypeScript
InstallLicense(LicenseFile: string): void;
```

```
C#
void InstallLicense(string LicenseFile);
```

```
C++
HRESULT InstallLicense(_bstr_t LicenseFile);
```

Parameters

`LicenseFile`

Full path to the license file.

Description

Install a given license file.

DisplayActivationDialog

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void DisplayActivationDialog(ulong WindowHandle);
```

```
C++
HRESULT DisplayActivationDialog(HWND WindowHandle);
```

Parameters

`WindowHandle`

Parent window handle.

Description

Displays the activation dialog and lets the user to manually activate the SPMC.

InstallLicenseInMemory

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void InstallLicenseInMemory(Array Data);
```

```
C++
HRESULT InstallLicenseInMemory(SAFEARRAY(byte) Data);
```

Parameters

`Data`

License data in memory as byte array.

Description

Install a license file, which contents is loaded into memory and provided in a byte array. Primary for managed clients.

InstallLicenseInMemoryNative

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT InstallLicenseInMemoryNative(const UCHAR * pData, ULONG Size);
```

Parameters

`pData`

Pointer to the license data in memory.

`Size`

License data size, in bytes.

Description

Install a given license file stored in a memory buffer. Provided primary for native clients who already loaded the license file into memory.

IMonitoring Interface

Description

This interface is implemented by the Monitor object in the Serial Monitoring Control library. You get this interface by calling the ISerialMonitor.CreateMonitor method and use it to start monitoring the serial device.

Declaration

```
TypeScript
interface IMonitoring {
    // Properties
    readonly ConnectedDevice: IDevice;
    readonly Connected: boolean;
    readonly SerialMonitor: ISerialMonitor;
    // Methods
    Connect(device?: IDevice | string, highPrecision = false): void;
    Disconnect(): void;
}
```

```
C#
public interface IMonitoring
{
    // Properties
    IDevice ConnectedDevice { get; }
    bool Connected { get; }
    ISerialMonitor SerialMonitor { get; }
    // Methods
    void Connect(object device, bool highPrecision = false);
    void Disconnect();
    void AddNativeListener(INativeListener listener);
    void RemoveNativeListener(INativeListener listener);
}
```

```
C++
struct IMonitoring : IDispatch
{
    // Properties
    IDevicePtr ConnectedDevice;  // get
    VARIANT_BOOL Connected;  // get
    ISerialMonitorPtr SerialMonitor;  // get
    // Methods
    HRESULT Connect(const _variant_t & device, VARIANT_BOOL highPrecision = VARIANT_FALSE);
    HRESULT Disconnect();
    HRESULT AddNativeListener(INativeListener * listener);
    HRESULT RemoveNativeListener(INativeListener * listener);
};
```

IMonitoring Properties

ConnectedDevice

```
TypeScript
readonly ConnectedDevice: IDevice;
```

```
C#
IDevice ConnectedDevice { get; }
```

```
C++
IDevicePtr ConnectedDevice;  // get
```

Description

Returns the device this monitor currently is connected to, or null if it is not connected.

Connected

```
TypeScript
readonly Connected: boolean;
```

```
C#
bool Connected { get; }
```

```
C++
VARIANT_BOOL Connected;  // get
```

Description

Returns `true` if this monitor object is currently connected to the serial device.

SerialMonitor

```
TypeScript
readonly SerialMonitor: ISerialMonitor;
```

```
C#
ISerialMonitor SerialMonitor { get; }
```

```
C++
ISerialMonitorPtr SerialMonitor;  // get
```

Description

Returns the reference to the main Serial Monitor object.

IMonitoring Methods

Connect

```
TypeScript
Connect(device?: IDevice | string, highPrecision = false): void;
```

```
C#
void Connect(object device, bool highPrecision = false);
```

```
C++
HRESULT Connect(const _variant_t & device, VARIANT_BOOL highPrecision = VARIANT_FALSE);
```

Parameters

`device`

This parameter may be one of the following:

String	Port name. This string name will be used as an index to the ISerialMonitor.Devices collection and the resulting IDevice object will be used
IDevice pointer	Pointer to the serial device you want to connect to
Omitted	Tell the monitor to wait for the next serial device to be plugged to the computer and connect to it immediately.

`highPrecision`

Set this parameter to true to use high-precision time and false otherwise.

Description

Tell the Monitor object to connect to the serial device. `device` can be the port name or the IDevice pointer. If omitted (`VT_EMPTY` variant in native code), the monitor will connect to the next connected serial device.

If you have already attached a Native Listener or an event sink to the monitor object, you will immediately receive the OnConnection event. If you omitted the `device` parameter, you would not receive the event until the new serial device is plugged.

Disconnect

```
TypeScript
Disconnect(): void;
```

```
C#
void Disconnect();
```

```
C++
HRESULT Disconnect();
```

Description

Disconnect the monitor object from the serial device.

AddNativeListener

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void AddNativeListener(INativeListener listener);
```

```
C++
HRESULT AddNativeListener(INativeListener * listener);
```

Parameters

`listener`

Pointer to the caller-implemented object with INativeListener interface.

Description

Adds new native listener to the Monitor object. This method should only be called by native clients.

RemoveNativeListener

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void RemoveNativeListener(INativeListener listener);
```

```
C++
HRESULT RemoveNativeListener(INativeListener * listener);
```

Parameters

`listener`

Pointer to the caller-implemented object with INativeListener interface.

Description

Removes the native listener from the list of registered native listeners. This method should only be called by native clients.

IDevice Interface

Description

This interface is implemented by the Serial Monitoring library for each registered serial device. Use this interface to query the serial device properties.

Declaration

```
TypeScript
interface IDevice {
    // Properties
    readonly Name: string;
    readonly ConnectionString: string;
    readonly Present: boolean;
    readonly Type: DeviceType;
    readonly Port: string;
    readonly Icon: number;
    readonly OpenedBy: boolean;
}
```

```
C#
public interface IDevice
{
    // Properties
    string Name { get; }
    string ConnectionString { get; }
    bool Present { get; }
    DeviceType Type { get; }
    string Port { get; }
    ulong Icon { get; }
    bool OpenedBy { get; }
}
```

```
C++
struct IDevice : IDispatch
{
    // Properties
    _bstr_t Name;  // get
    _bstr_t ConnectionString;  // get
    VARIANT_BOOL Present;  // get
    DeviceType Type;  // get
    _bstr_t Port;  // get
    HICON Icon;  // get
    VARIANT_BOOL OpenedBy;  // get
};
```

IDevice Properties

Name

```
TypeScript
readonly Name: string;
```

```
C#
string Name { get; }
```

```
C++
_bstr_t Name;  // get
```

Description

This property returns the string containing the device name. This name can be used only for identification purposes and shows the driver-assigned name for the serial device.

ConnectionString

```
TypeScript
readonly ConnectionString: string;
```

```
C#
string ConnectionString { get; }
```

```
C++
_bstr_t ConnectionString;  // get
```

Description

Returns the device connection string. This is the preferred way to work with the serial devices in the native code. If you need to open the device in your code (instead of monitoring it), find the device using the Devices collection, and query its `ConnectionString` property. Then pass the returned string to the `CreateFile` Windows API function to get the handle to the device.

Present

```
TypeScript
readonly Present: boolean;
```

```
C#
bool Present { get; }
```

```
C++
VARIANT_BOOL Present;  // get
```

Description

The Serial Monitoring Control's devices collection always contains all serial devices registered in your system. Some of the devices may not be physically present or may be disabled in the Device Manager. However those devices are still present in the Devices collection, but the `Present` property is `false` for them. It is an error to try to attach the monitor object to the non-present device.

Type

```
TypeScript
readonly Type: DeviceType;
```

```
C#
DeviceType Type { get; }
```

```
C++
DeviceType Type;  // get
```

Description

This property contains the device's type. See the DeviceType enumeration for a list of possible values.

Port

```
TypeScript
readonly Port: string;
```

```
C#
string Port { get; }
```

```
C++
_bstr_t Port;  // get
```

Description

Port name.

> **NOTE**
>
> For several kinds of serial devices this property returns an empty string. In this case use the ConnectionString property to get the correct device address.

Icon

```
TypeScript
readonly Icon: number;
```

```
C#
ulong Icon { get; }
```

```
C++
HICON Icon;  // get
```

Description

Use the returned icon handle to display the device class image. The icon is a system-defined icon for a device class (port or modem). You **should not** destroy the returned handle with a call to `DestroyObject` function.

OpenedBy

```
TypeScript
readonly OpenedBy: boolean;
```

```
C#
bool OpenedBy { get; }
```

```
C++
VARIANT_BOOL OpenedBy;  // get
```

Description

The Serial Monitoring Control lets you determine the name and process identifier of the process currently having the serial device opened. This property contains the name of the process which currently has the device opened. The returned string is in form `processname (processid)` where the `processname` is the name of the executable file (without extension) and `processid` is the numeric process identifier. Both values are the same as displayed by the Windows Task Manager. If the device is not currently opened by any process, the returned string is empty.

IDeviceCollection Interface

Description

This interface is used to enumerate existing serial devices. It is implemented by the Serial Monitoring Control.

You obtain this interface by taking the value of the ISerialMonitor.Devices property and use it to enumerate the installed serial devices. There are two ways for using this interface. You can get the Count property value to get the number of devices in the collection and then use the Item property to get the IDevice interface for each device in a collection.

Another way of enumerating the devices in the collection is to take the value of the _NewEnum property to get the object exposing the `IEnumVARIANT` interface and use its properties and methods to enumerate the collection.

Note that usually this process is somehow automated in scripting and CLR languages.

Declaration

```
TypeScript
// This interface is not available in scripting environment
```

```
C#
public interface IDeviceCollection
{
    // Properties
    IDevice this[object Index] { get; set; }
    int Count { get; }
    // Methods
    IEnumerator GetEnumerator();
}
```

```C++
struct IDeviceCollection : IDispatch
{
    // Properties
    IDevicePtr Item(const _variant_t & Index);  // get set
    long Count;  // get
    // Methods
    IUnknownPtr Get_NewEnum();
};
```

IDeviceCollection Properties

this

```TypeScript
// This property is not available in scripting environment
```

```C#
IDevice this[object Index] { get; set; }
```

```C++
IDevicePtr Item(const _variant_t & Index);  // get set
```

Description

Returns the device from the collection.

Count

```TypeScript
// This property is not available in scripting environment
```

```C#
int Count { get; }
```

```C++
long Count;  // get
```

Description

Returns the total number of devices in a collection object.

IDeviceCollection Methods

GetEnumerator

```TypeScript
// This method is not available in scripting environment
```

```C#
IEnumerator GetEnumerator();
```

```C++
IUnknownPtr Get_NewEnum();
```

Description

Returns standard OLE enumerator for this object.

IDevice

INativeListener Interface

Description

The high-performance event interface for native (C/C++) listeners.

This is a local interface, which means that it can only be used by the in-proc binary compatible clients, written, for example, in unmanaged C++ language.

After you first register your native listener with a call to IMonitoring.AddNativeListener method call, your native listener is queried for its capabilities through the call to the INativeListener.GetCaps method. If it returns `ACCEPT_RAW` (value is `1`) value, the Serial Monitoring Control library will call the INativeListener.ProcessRAWBuffer with a pointer to a raw buffer. The raw buffer contains unparsed monitored data. This is the fastest method of getting data from the Serial Monitoring Control library. Using this method, you will be able to match the performance of the HHD Software Serial Monitor application. Note, that the raw listeners never get their `OnXXX` methods called. All data is passed to the `ProcessRAWBuffer` method and you must parse the data in your code. If you want the control to parse data for you, do not return `ACCEPT_RAW` value from the `GetCaps` method.

If your `GetCaps` method returns zero, the control will parse the monitored data and call one of the following `OnXXX` methods, described below. For performance reasons, if one of the methods ever returns `E_NOTIMPL` error code, the control will stop parsing the corresponding request type and stop calling not implemented methods for your listener object. If you have multiple listeners registered for the single monitoring object, and one of your listeners returns E_NOTIMPL for one of the events, the control still continues processing and calling the method for other registered listeners.

Note, that the `GetCaps` method is called immediately after you register your listener. The control will not call the method anymore, so you cannot change the type of the listener once it has been determined.

Declaration

TypeScript

```
// This interface is not available in scripting environment
```

C#

```
// This interface is not available in managed environment
```

C++

```
struct INativeListener : IDispatch
{
    // Methods
    DWORD GetCaps();
    HRESULT ProcessRAWBuffer(void * pData, DWORD Size);
    HRESULT OnConnection(FILETIME * fTime, BOOL bConnected, LPCTSTR DeviceName);
    HRESULT OnOpen(FILTIME * fTime, LPCTSTR ProcessName, ULONG ProcessId);
    HRESULT OnClosed(FILETIME * fTime);
    HRESULT OnRead(FILETIME * fTime, void * pData, ULONG Size);
    HRESULT OnWrite(FILETIME * fTime, void * pData, ULONG Size);
    HRESULT OnTransmit(FILETIME * fTime, void * pData, ULONG Size);
    HRESULT OnBaudRate(FILETIME * fTime, ULONG BaudRate, BOOL bGet);
    HRESULT OnSerialChars(FILETIME * fTime,
        UCHAR Eof,
        UCHAR Error,
        UCHAR Break,
        UCHAR Event,
        UCHAR Xon,
        UCHAR Xoff,
        BOOL bGet);
    HRESULT OnCommStatus(FILETIME * fTime,
        ULONG Errors,
```

```
        ULONG HoldReasons,
        ULONG AmountInInQueue,
        ULONG AmountInOutQueue,
        BOOL EofReceived,
        BOOL WaitForImmediate);
    HRESULT OnDTRRTS(FILETIME * fTime, BOOL DTR, BOOL RTS);
    HRESULT OnHandflow(FILETIME * fTime,
        ULONG ControlHandShake,
        ULONG FlowReplace,
        ULONG XonLimit,
        ULONG XoffLimit,
        BOOL bGet);
    HRESULT OnLineControl(FILETIME * fTime,
        ULONG WordLength,
        STOPBITS StopBits,
        PARITY Parity,
        BOOL bGet);
    HRESULT OnStats(FILETIME * fTime,
        ULONG ReceivedCount,
        ULONG TransmittedCount,
        ULONG FrameErrorCount,
        ULONG SerialOverrunErrorCount,
        ULONG BufferOverrunErrorCount,
        ULONG ParityErrorCount);
    HRESULT OnTimeouts(FILETIME * fTime,
        ULONG ReadIntervalTimeout,
        ULONG ReadTotalTimeoutMultiplier,
        ULONG ReadTotalTimeoutConstant,
        ULONG WriteTotalTimeoutMultiplier,
        ULONG WriteTotalTimeoutConstant,
        BOOL bGet);
    HRESULT OnWaitMask(FILETIME * fTime, EVENTS WaitMask, BOOL bGet);
    HRESULT OnPurge(FILETIME * fTime, PURGE Purge);
    HRESULT OnSetQueueSize(FILETIME * fTime, ULONG InSize, ULONG OutSize);
    HRESULT OnSetBreak(FILETIME * fTime, BOOL bOn);
    HRESULT OnDTR(FILETIME * fTime, BOOL bSet);
    HRESULT OnReset(FILETIME * fTime);
    HRESULT OnRTS(FILETIME * fTime, BOOL bSet);
    HRESULT OnXOFF(FILETIME * fTime);
    HRESULT OnXON(FILETIME * fTime);
    HRESULT OnWaitOnMask(FILETIME * fTime, EVENTS WaitMask);
    HRESULT OnGetModemStatus(FILETIME * fTime, MODEMSTATUS ModemStatus);
    HRESULT OnGetProperties(FILETIME * fTime,
        USHORT PacketLength,
        USHORT PacketVersion,
        ULONG MaxTxQueue,
        ULONG MaxRxQueue,
        ULONG CurrentTxQueue,
        ULONG CurrentRxQueue,
        BAUDRATES MaxBaudRate,
        PROVIDERTYPE ProviderType,
        PROVIDERCAPS ProviderCaps,
        PROVIDERSETTABLE ProviderSettableParams,
        BAUDRATES SettableBaudRates,
        DATABITS_ST SettableData,
        STOPPARITY_ST SettableStopParity);
    HRESULT OnClearStats(FILETIME * fTime);
};
```

INativeListener Methods

GetCaps

TypeScript

```
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
DWORD GetCaps();
```

Description

Return the listener capabilities to the Monitor. If your listener returns `ACCEPT_RAW` value, it is considered the raw listener. The library will not parse the monitored data for a raw listener, instead, it will pass all data (buffered) to the INativeListener.ProcessRAWBuffer method.

This method is called only once.

ProcessRAWBuffer

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT ProcessRAWBuffer(void * pData, DWORD Size);
```

Parameters

`pData`

Pointer to a raw data.

`Size`

The size of the buffer pointed to by the `pData` parameter.

Description

Called by the control to process the monitored events at the lowest possible level. You will find the documentation for the raw processing in a separate topic.

OnConnection

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnConnection(FILETIME * fTime, BOOL bConnected, LPCTSTR DeviceName);
```

Parameters

`fTime`

The time of the event.

`bConnected`

The connection state. Indicates whether the monitor is connecting to the device, or disconnecting from it. See the ConnectionState enumeration for more information.

`DeviceName`

Pointer to the null-terminated name of the serial device the monitor is connecting to (disconnecting from).

Description

Called when the control attaches/detaches itself to/from the serial device.

OnOpen

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnOpen(FILTIME * fTime, LPCTSTR ProcessName, ULONG ProcessId);
```

Parameters

`fTime`

The time of the event.

`ProcessName`

Pointer to the null-terminated string containing the name of the process opening the device.

`ProcessId`

The process identifier.

Description

Called when the device is opened by monitored application.

OnClosed

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnClosed(FILETIME * fTime);
```

Parameters

`fTime`

The time of the event.

Description

Called when the device is closed by monitored application.

OnRead

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnRead(FILETIME * fTime, void * pData, ULONG Size);
```

Parameters

`fTime`

The time of the event.

`pData`

Pointer to the data read by the application in one function call.

`Size`

The size of the buffer pointed to by the `pData` parameter.

Description

Called when the monitored application is reading data.

OnWrite

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnWrite(FILETIME * fTime, void * pData, ULONG Size);
```

Parameters

`fTime`

The time of the event.

`pData`

Pointer to the data written by the application in one function call.

`Size`

The size of the buffer pointed to by the `pData` parameter.

Description

Called when the monitored application is writing data.

OnTransmit

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnTransmit(FILETIME * fTime, void * pData, ULONG Size);
```

Parameters

`fTime`

The time of the event.

`pData`

Pointer to the data transmitted by the application in one function call.

`Size`

The size of the buffer pointed to by the `pData` parameter.

Description

Called when the monitored application is transmitting data.

OnBaudRate

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnBaudRate(FILETIME * fTime, ULONG BaudRate, BOOL bGet);
```

Parameters

`fTime`

The time of the event.

`BaudRate`

Baud rate to set or get.

`bGet`

`TRUE` if application is reading current baud rate, or `FALSE` if it is changing it.

Description

Called when the monitored application retrieves/sets current baud rate.

OnSerialChars

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnSerialChars(FILETIME * fTime,
    UCHAR Eof,
    UCHAR Error,
    UCHAR Break,
    UCHAR Event,
    UCHAR Xon,
    UCHAR Xoff,
    BOOL bGet);
```

Parameters

`fTime`

The time of the event.

`Eof`

EOF character.

`Error`

ERROR character.

`Break`

BREAK character.

`Event`

EVENT character.

`Xon`

XON character.

`Xoff`

XOFF character.

`bGet`

`TRUE` if application is reading special characters, or `FALSE` if it is changing them.

Description

Called when the monitored application retrieves/sets special serial characters.

OnCommStatus

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnCommStatus(FILETIME * fTime,
    ULONG Errors,
    ULONG HoldReasons,
    ULONG AmountInInQueue,
    ULONG AmountInOutQueue,
    BOOL EofReceived,
    BOOL WaitForImmediate);
```

Parameters

`fTime`

The time of the event.

`Errors`

The number of errors for the port.

`HoldReasons`

The number of hold reasons on the port.

`AmountInInQueue`

Amount of data in the input queue, in bytes.

`AmountInOutQueue`

Amount of data in the output queue, in bytes.

`EofReceived`

`TRUE` if the `EOF` has been received.

`WaitForImmediate`

`TRUE` if the port is in wait state.

Description

Called when the monitored application reads the comm port status.

OnDTRRTS

TypeScript

```
// This method is not available in scripting environment
```

C#

```
// This method is not available in managed environment
```

C++

```
HRESULT OnDTRRTS(FILETIME * fTime, BOOL DTR, BOOL RTS);
```

Parameters

`fTime`

The time of the event.

`DTR`

DTR line state.

`RTS`

RTS line state.

Description

Called when the monitored application retrieves DTR and RTS line states.

OnHandflow

TypeScript

```
// This method is not available in scripting environment
```

C#

```
// This method is not available in managed environment
```

```
C++
HRESULT OnHandflow(FILETIME * fTime,
    ULONG ControlHandShake,
    ULONG FlowReplace,
    ULONG XonLimit,
    ULONG XoffLimit,
    BOOL bGet);
```

Parameters

`fTime`

The time of the event.

`ControlHandShake`

ControlHandShake

`FlowReplace`

FlowReplace

`XonLimit`

XonLimit

`XoffLimit`

XoffLimit

`bGet`

`TRUE` if the application reads the handshake information or `FALSE` otherwise.

OnLineControl

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnLineControl(FILETIME * fTime,
    ULONG WordLength,
    STOPBITS StopBits,
    PARITY Parity,
    BOOL bGet);
```

Parameters

`fTime`

The time of the event.

`WordLength`

Byte size.

`StopBits`

Stop bits. One of the values from the STOPBITS enumeration.

`Parity`

Parity. One of the values from the PARITY enumeration.

`bGet`

`TRUE` if the application reads the line control settings, `FALSE` if writes it.

Description

Called when the monitored application retrieves/sets the line control options (such as stop bits, parity and word length).

OnStats

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnStats(FILETIME * fTime,
    ULONG ReceivedCount,
    ULONG TransmittedCount,
    ULONG FrameErrorCount,
    ULONG SerialOverrunErrorCount,
    ULONG BufferOverrunErrorCount,
    ULONG ParityErrorCount);
```

Parameters

`fTime`

The time of the event.

`ReceivedCount`

The number of received bytes.

`TransmittedCount`

The number of transmitted bytes.

`FrameErrorCount`

The number of frame errors.

`SerialOverrunErrorCount`

The number of serial overrun errors.

`BufferOverrunErrorCount`

The number of buffer overrun errors.

`ParityErrorCount`

The number of parity errors.

Description

Called when the monitored application reads port statistics.

OnTimeouts

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnTimeouts(FILETIME * fTime,
    ULONG ReadIntervalTimeout,
    ULONG ReadTotalTimeoutMultiplier,
    ULONG ReadTotalTimeoutConstant,
    ULONG WriteTotalTimeoutMultiplier,
    ULONG WriteTotalTimeoutConstant,
    BOOL bGet);
```

Parameters

`fTime`

The time of the event.

`ReadIntervalTimeout`

Read interval timeout.

`ReadTotalTimeoutMultiplier`

Read total timeout multiplier.

`ReadTotalTimeoutConstant`

Read total timeout constant.

`WriteTotalTimeoutMultiplier`

Write total timeout multiplier.

`WriteTotalTimeoutConstant`

Write total timeout constant.

`bGet`

`TRUE` if the application reads this information or `FALSE` otherwise.

Description

Called when the monitored application retrieves/sets port timeouts.

OnWaitMask

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnWaitMask(FILETIME * fTime, EVENTS WaitMask, BOOL bGet);
```

Parameters

`fTime`

The time of the event.

`WaitMask`

The wait mask. See the EVENTS enumeration.

`bGet`

`TRUE` if the monitored application reads the wait mask or `FALSE` otherwise.

Description

Called when the monitored application retrieves/sets the wait mask.

OnPurge

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnPurge(FILETIME * fTime, PURGE Purge);
```

Parameters

`fTime`

The time of the event.

`Purge`

The bit mask indicating what data to purge. See the PURGE enumeration for more information.

Description

Called when the monitored application purges the port.

OnSetQueueSize

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnSetQueueSize(FILETIME * fTime, ULONG InSize, ULONG OutSize);
```

Parameters

`fTime`

The time of the event.

`InSize`

The size of the input queue.

`OutSize`

The size of the output queue.

Description

Called when the monitored application sets queue sizes.

OnSetBreak

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnSetBreak(FILETIME * fTime, BOOL bOn);
```

Parameters

`fTime`

The time of the event.

`bOn`

`TRUE` if application sets the break signal, or `FALSE` otherwise.

Description

Called when the monitored application sets/resets the break signal.

OnDTR

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnDTR(FILETIME * fTime, BOOL bSet);
```

Parameters

`fTime`

The time of the event.

`bSet`

`TRUE` if the application sets the DTR line state, or `FALSE` if the application resets it.

Description

Called when the monitored application sets/resets DTR line state.

OnReset

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnReset(FILETIME * fTime);
```

Parameters

`fTime`

The time of the event.

Description

Called when the monitored application resets the port.

OnRTS

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnRTS(FILETIME * fTime, BOOL bSet);
```

Parameters

`fTime`

The time of the event.

`bSet`

`TRUE` if the application sets the RTS line state, or `FALSE` if the application resets it.

Description

Called when the monitored application sets/resets RTS line state.

OnXOFF

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnXOFF(FILETIME * fTime);
```

Parameters

`fTime`

The time of the event.

Description

Called when the monitored application sends the XOFF character.

OnXON

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnXON(FILETIME * fTime);
```

Parameters

`fTime`

The time of the event.

Description

Called when the monitored application sends the XON character.

OnWaitOnMask

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnWaitOnMask(FILETIME * fTime, EVENTS WaitMask);
```

Parameters

`fTime`

The time of the event.

`WaitMask`

The wait mask to wait. See the EVENTS enumeration for more information.

Description

Called when the monitored application waits on mask.

OnGetModemStatus

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnGetModemStatus(FILETIME * fTime, MODEMSTATUS ModemStatus);
```

Parameters

`fTime`

The time of the event.

`ModemStatus`

The modem status is returned as a combination of flags in the MODEMSTATUS enumeration.

Description

Called when the monitored application reads the modem status.

OnGetProperties

```
TypeScript
// This method is not available in scripting environment
```

```
C#
// This method is not available in managed environment
```

```
C++
HRESULT OnGetProperties(FILETIME * fTime,
    USHORT PacketLength,
    USHORT PacketVersion,
    ULONG MaxTxQueue,
    ULONG MaxRxQueue,
    ULONG CurrentTxQueue,
    ULONG CurrentRxQueue,
    BAUDRATES MaxBaudRate,
    PROVIDERTYPE ProviderType,
    PROVIDERCAPS ProviderCaps,
    PROVIDERSETTABLE ProviderSettableParams,
    BAUDRATES SettableBaudRates,
    DATABITS_ST SettableData,
    STOPPARITY_ST SettableStopParity);
```

Parameters

`fTime`

The time of the event.

`PacketLength`

The packet length, in bytes.

`PacketVersion`

The packet version.

`MaxTxQueue`

Maximum transmit queue size, in bytes.

`MaxRxQueue`

Maximum receive queue size, in bytes.

`CurrentTxQueue`

Current transmit queue size, in bytes.

`CurrentRxQueue`

Current receive queue size, in bytes.

`MaxBaudRate`

Maximum baud rate.

`ProviderType`

Provider type. See the PROVIDERTYPE enumeration for more information.

`ProviderCaps`

Provider capabilities. See the PROVIDERCAPS enumeration for more information.

`ProviderSettableParams`

Provider settable parameters. Indicates what port parameters may be set for a given device. See the PROVIDERSETTABLE enumeration for more information.

`SettableBaudRates`

The bit mask indicating what baud rates are supported for a given device. See the BAUDRATES enumeration for more information.

`SettableData`

The bit mask indicating what data bits are supported for a given device. See the DATABITS_ST enumeration for more information.

`SettableStopParity`

The bit mask indicating what stop bits and parity are supported for a given device. See the STOPPARITY_ST enumeration for more information.

Description

Called when the monitored application retrieves the comm properties.

OnClearStats

TypeScript

```
// This method is not available in scripting environment
```

C#

```
// This method is not available in managed environment
```

C++

```
HRESULT OnClearStats(FILETIME * fTime);
```

Parameters

`fTime`

The time of the event.

Description

Called when the monitored application clears the statistics.

_ISerialMonitorEvents Interface

Description

You implement this interface to receive events fired by the Serial Monitor Control library's main object. The Serial Monitor Control calls the_ISerialMonitorEvents.OnChange method when there is a change in the device collection entries. Please see documentation for the _IMonitoringEvents interface for information on binding to the event source.

Declaration

TypeScript

```
// This interface is not available in scripting environment
```

C#

```
public interface _ISerialMonitorEvents
{
    // Methods
    void OnChange();
}
```

```
C++
struct _ISerialMonitorEvents : IDispatch
{
    // Methods
    HRESULT OnChange();
};
```

_ISerialMonitorEvents Methods

OnChange

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnChange();
```

```
C++
HRESULT OnChange();
```

Description

Fired when device appears or disappears.

_IMonitoringEvents Interface

Description

An event source interface for managed and scripting clients. You do not explicitly implement this interface. It is usually implemented by the language runtime. You register the so-called "events", one for each method of this interface to handle specific monitored requests.

This interface is used by managed clients. The language runtime usually uses the methods of this interface automatically, allowing the client to register events, callbacks or delegates, which are called when the Serial Monitoring Control library fires these events. There is a proprietary interface in each managed language to connect to event sources. You will see Microsoft C# examples in this documentation, for other languages, please consult their documentation for a proper syntax to handle events.

Declaration

```
TypeScript
// This interface is not available in scripting environment
```

C#

```
public interface _IMonitoringEvents
{
    // Methods
    void OnConnection(DateTime time, ConnectionState cs, string Name);
    void OnOpen(DateTime time, string Name, uint ProcessId);
    void OnClose(DateTime time);
    void OnRead(DateTime time, Array array);
    void OnWrite(DateTime time, Array array);
    void OnTransmit(DateTime time, Array array);
    void OnBaudRate(DateTime time, uint BaudRate, bool bGet);
    void OnSerialChars(DateTime fTime, byte Eof, byte Error, byte Break, byte Event, byte Xon
    void OnCommStatus(DateTime fTime,
        uint Errors,
        uint HoldReasons,
        uint AmountInInQueue,
        uint AmountInOutQueue,
        bool EofReceived,
        bool WaitForImmediate);
    void OnDTRRTS(DateTime fTime, bool DTR, bool RTS);
    void OnHandflow(DateTime fTime, uint ControlHandShake, uint FlowReplace, uint XonLimit, u
    void OnLineControl(DateTime fTime, uint WordLength, STOPBITS StopBits, PARITY Parity, boo
    void OnStats(DateTime fTime,
        uint ReceivedCount,
        uint TransmittedCount,
        uint FrameErrorCount,
        uint SerialOverrunErrorCount,
        uint BufferOverrunErrorCount,
        uint ParityErrorCount);
    void OnTimeouts(DateTime fTime,
        uint ReadIntervalTimeout,
        uint ReadTotalTimeoutMultiplier,
        uint ReadTotalTimeoutConstant,
        uint WriteTotalTimeoutMultiplier,
        uint WriteTotalTimeoutConstant,
        bool bGet);
    void OnWaitMask(DateTime fTime, EVENTS WaitMask, bool bGet);
    void OnPurge(DateTime fTime, PURGE Purge);
    void OnSetQueueSize(DateTime fTime, uint InSize, uint OutSize);
    void OnSetBreak(DateTime fTime, bool bOn);
    void OnDTR(DateTime fTime, bool bSet);
    void OnReset(DateTime fTime);
    void OnRTS(DateTime fTime, bool bSet);
    void OnXOFF(DateTime fTime);
    void OnXON(DateTime fTime);
    void OnWaitOnMask(DateTime fTime, EVENTS WaitMask);
    void OnGetModemStatus(DateTime fTime, MODEMSTATUS ModemStatus);
    void OnGetProperties(DateTime fTime,
        ushort PacketLength,
        ushort PacketVersion,
        uint MaxTxQueue,
        uint MaxRxQueue,
        uint CurrentTxQueue,
        uint CurrentRxQueue,
        BAUDRATES MaxBaudRate,
        PROVIDERTYPE ProviderType,
        PROVIDERCAPS ProviderCaps,
        PROVIDERSETTABLE ProviderSettableParams,
        BAUDRATES SettableBaudRates,
        DATABITS_ST SettableData,
        STOPPARITY_ST SettableStopParity);
    void OnClearStats(DateTime fTime);
}
```

C++

```
// This interface is not available in native environment
```

_IMonitoringEvents Methods

OnConnection

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnConnection(DateTime time, ConnectionState cs, string Name);
```

```
C++
// This method is not available in native environment
```

Parameters

`time`

The time of the event.

`cs`

The connection state. Indicates whether the monitor is connecting to the device, or disconnecting from it. See the ConnectionState enumeration for more information.

`Name`

The name of the serial device the monitor is connecting to (disconnecting from).

Description

Fired when the monitor object attaches/detaches itself to/from the serial device.

OnOpen

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnOpen(DateTime time, string Name, uint ProcessId);
```

```
C++
// This method is not available in native environment
```

Parameters

`time`

The time of the event.

`Name`

String containing the name of the process opening the device.

`ProcessId`

The process identifier.

Description

Fired when the device is opened by the monitored application.

OnClose

```TypeScript
// This method is not available in scripting environment
```

```C#
void OnClose(DateTime time);
```

```C++
// This method is not available in native environment
```

Parameters

`time`

The time of the event.

Description

Fired when the device is closed by the monitored application.

OnRead

```TypeScript
// This method is not available in scripting environment
```

```C#
void OnRead(DateTime time, Array array);
```

```C++
// This method is not available in native environment
```

Parameters

`time`

The time of the event.

`array`

The data array. Contains the data read by the application in one function call. This parameter is a byte array.

Description

Called when the monitored application is reading data.

Example

Binding the `OnRead` event in C#:

```C#
// Binding the event handler
monitor.OnRead += (time, array)=>
{
 var data = (byte[]) array; // cast the object array to the byte array
 for (int i = 0; i < data.Length; i++) // read and process data
  ProcessCharacter(data[i]);
};
```

OnWrite

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnWrite(DateTime time, Array array);
```

```
C++
// This method is not available in native environment
```

Parameters

`time`

The time of the event.

`array`

The data array. Contains the data written by the application in one function call. This parameter is a byte array.

Description

Called when the monitored application is writing data.

OnTransmit

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnTransmit(DateTime time, Array array);
```

```
C++
// This method is not available in native environment
```

Parameters

`time`

The time of the event.

`array`

The data array. Contains the data transmitted by the application in one function call. This parameter is a byte array.

Description

Called when the monitored application is transmitting data.

OnBaudRate

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnBaudRate(DateTime time, uint BaudRate, bool bGet);
```

```
C++
// This method is not available in native environment
```

Parameters

`time`

The time of the event.

`BaudRate`

The baud rate to set or get.

`bGet`

`true` if application is reading current baud rate, or `false` if it is changing it.

Description

Called when the monitored application retrieves/sets current baud rate.

OnSerialChars

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnSerialChars(DateTime fTime, byte Eof, byte Error, byte Break, byte Event, byte Xon, by
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`Eof`

EOF character.

`Error`

ERROR character.

`Break`

BREAK character.

`Event`

EVENT character.

`Xon`

XON character.

`Xoff`

XOFF character.

`bGet`

`true` if application is reading special characters, or `false` if it is changing them.

Description

Fired when the monitored application retrieves/sets special serial characters.

OnCommStatus

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnCommStatus(DateTime fTime,
    uint Errors,
    uint HoldReasons,
    uint AmountInInQueue,
    uint AmountInOutQueue,
    bool EofReceived,
    bool WaitForImmediate);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`Errors`

The number of errors for the port.

`HoldReasons`

The number of hold reasons on the port.

`AmountInInQueue`

Amount of data in the input queue, in bytes.

`AmountInOutQueue`

Amount of data in the output queue, in bytes.

`EofReceived`

`true` if the `EOF` has been received.

`WaitForImmediate`

`true` if the port is in wait state.

Description

Fired when the monitored application reads the comm port status.

OnDTRRTS

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnDTRRTS(DateTime fTime, bool DTR, bool RTS);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`DTR`

DTR line state.

`RTS`

RTS line state.

Description

Fired when the monitored application retrieves DTR and RTS line states.

OnHandflow

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnHandflow(DateTime fTime, uint ControlHandShake, uint FlowReplace, uint XonLimit, uint
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`ControlHandShake`

ControlHandShake

`FlowReplace`

FlowReplace

`XonLimit`

XonLimit

`XoffLimit`

XoffLimit

`bGet`

`true` if the application reads the handshake information or `false` otherwise.

OnLineControl

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnLineControl(DateTime fTime, uint WordLength, STOPBITS StopBits, PARITY Parity, bool bG
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`WordLength`

Byte size.

`StopBits`

Stop bits. One of the values from the STOPBITS enumeration.

`Parity`

Parity. One of the values from the PARITY enumeration.

`bGet`

`true` if the application reads the line control settings, `false` if writes it.

Description

Fired when the monitored application retrieves/sets the line control options (such as stop bits, parity and word length).

OnStats

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnStats(DateTime fTime,
    uint ReceivedCount,
    uint TransmittedCount,
    uint FrameErrorCount,
    uint SerialOverrunErrorCount,
    uint BufferOverrunErrorCount,
    uint ParityErrorCount);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`ReceivedCount`

The number of received bytes.

`TransmittedCount`

The number of transmitted bytes.

`FrameErrorCount`

The number of frame errors.

`SerialOverrunErrorCount`

The number of serial overrun errors.

`BufferOverrunErrorCount`

The number of buffer overrun errors.

`ParityErrorCount`

The number of parity errors.

Description

Fired when the monitored application reads port statistics.

OnTimeouts

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnTimeouts(DateTime fTime,
    uint ReadIntervalTimeout,
    uint ReadTotalTimeoutMultiplier,
    uint ReadTotalTimeoutConstant,
    uint WriteTotalTimeoutMultiplier,
    uint WriteTotalTimeoutConstant,
    bool bGet);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`ReadIntervalTimeout`

Read interval timeout.

`ReadTotalTimeoutMultiplier`

Read total timeout multiplier.

`ReadTotalTimeoutConstant`

Read total timeout constant.

`WriteTotalTimeoutMultiplier`

Write total timeout multiplier.

`WriteTotalTimeoutConstant`

Write total timeout constant.

`bGet`

`true` if the application reads this information or `false` otherwise.

Description

Fired when the monitored application retrieves/sets port timeouts.

OnWaitMask

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnWaitMask(DateTime fTime, EVENTS WaitMask, bool bGet);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`WaitMask`

The wait mask. See the EVENTS enumeration.

`bGet`

`true` if the monitored application reads the wait mask or `false` otherwise.

Description

Fired when the monitored application retrieves/sets the wait mask.

OnPurge

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnPurge(DateTime fTime, PURGE Purge);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`Purge`

The bit mask indicating what data to purge. See the PURGE enumeration for more information.

Description

Fired when the monitored application purges the port.

OnSetQueueSize

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnSetQueueSize(DateTime fTime, uint InSize, uint OutSize);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`InSize`

The size of the input queue.

`OutSize`

The size of the output queue.

Description

Fired when the monitored application sets queue sizes.

OnSetBreak

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnSetBreak(DateTime fTime, bool bOn);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`bOn`

`true` if application sets the break signal, or `false` otherwise.

Description

Fired when the monitored application sets/resets the break signal.

OnDTR

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnDTR(DateTime fTime, bool bSet);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`bSet`

`true` if the application sets the DTR line state, or `false` if the application resets it.

Description

Fired when the monitored application sets/resets DTR line state.

OnReset

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnReset(DateTime fTime);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

Description

Fired when the monitored application resets the port.

OnRTS

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnRTS(DateTime fTime, bool bSet);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`bSet`

`true` if the application sets the RTS line state, or `false` if the application resets it.

Description

Fired when the monitored application sets/resets RTS line state.

OnXOFF

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnXOFF(DateTime fTime);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

Description

Fired when the monitored application sends the XOFF character.

OnXON

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnXON(DateTime fTime);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

Description

Fired when the monitored application sends the XON character.

OnWaitOnMask

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnWaitOnMask(DateTime fTime, EVENTS WaitMask);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`WaitMask`

The wait mask to wait. See the EVENTS enumeration for more information.

Description

Fired when the monitored application waits on mask.

OnGetModemStatus

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnGetModemStatus(DateTime fTime, MODEMSTATUS ModemStatus);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`ModemStatus`

The modem status is returned as a combination of flags in the MODEMSTATUS enumeration.

Description

Fired when the monitored application reads the modem status.

OnGetProperties

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnGetProperties(DateTime fTime,
    ushort PacketLength,
    ushort PacketVersion,
    uint MaxTxQueue,
    uint MaxRxQueue,
    uint CurrentTxQueue,
    uint CurrentRxQueue,
    BAUDRATES MaxBaudRate,
    PROVIDERTYPE ProviderType,
    PROVIDERCAPS ProviderCaps,
    PROVIDERSETTABLE ProviderSettableParams,
    BAUDRATES SettableBaudRates,
    DATABITS_ST SettableData,
    STOPPARITY_ST SettableStopParity);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

`PacketLength`

The packet length, in bytes.

`PacketVersion`

The packet version.

`MaxTxQueue`

Maximum transmit queue size, in bytes.

`MaxRxQueue`

Maximum receive queue size, in bytes.

`CurrentTxQueue`

Current transmit queue size, in bytes.

`CurrentRxQueue`

Current receive queue size, in bytes.

`MaxBaudRate`

Maximum baud rate.

`ProviderType`

Provider type. See the PROVIDERTYPE enumeration for more information.

`ProviderCaps`

Provider capabilities. See the PROVIDERCAPS enumeration for more information.

`ProviderSettableParams`

Provider settable parameters. Indicates what port parameters may be set for a given device. See the PROVIDERSETTABLE enumeration for more information.

`SettableBaudRates`

The bit mask indicating what baud rates are supported for a given device. See the BAUDRATES enumeration for more information.

`SettableData`

The bit mask indicating what data bits are supported for a given device. See the DATABITS_ST enumeration for more information.

`SettableStopParity`

The bit mask indicating what stop bits and parity are supported for a given device. See the STOPPARITY_ST enumeration for more information.

Description

Fired when the monitored application retrieves the comm properties.

OnClearStats

```
TypeScript
// This method is not available in scripting environment
```

```
C#
void OnClearStats(DateTime fTime);
```

```
C++
// This method is not available in native environment
```

Parameters

`fTime`

The time of the event.

Description

Fired when the monitored application clears the statistics.

DeviceType Enumeration

Symbol	Value	Description
`DEVICETYPE_UNKNOWN`	`0x00000000`	The type of the device is unknown.
`DEVICETYPE_PORT`	`0x00000001`	The device is a standard serial communication port.
`DEVICETYPE_MODEM`	`0x00000002`	The device is a standard modem.
`DEVICETYPE_BOTH`	`0x00000003`	The device behaves as a communication port and as a modem.

BAUDRATES Enumeration

Description

Baud rate settable parameters.

Symbol	Value	Description
HHD_BAUD_075	0x00000001	75 bps
HHD_BAUD_110	0x00000002	110 bps
HHD_BAUD_134_5	0x00000004	134.5 bps
HHD_BAUD_150	0x00000008	150 bps
HHD_BAUD_300	0x00000010	300 bps
HHD_BAUD_600	0x00000020	600 bps
HHD_BAUD_1200	0x00000040	1200 bps
HHD_BAUD_1800	0x00000080	1800 bps
HHD_BAUD_2400	0x00000100	2400 bps
HHD_BAUD_4800	0x00000200	4800 bps
HHD_BAUD_7200	0x00000400	7200 bps
HHD_BAUD_9600	0x00000800	9600 bps
HHD_BAUD_14400	0x00001000	14400 bps
HHD_BAUD_19200	0x00002000	19200 bps
HHD_BAUD_38400	0x00004000	38400 bps
HHD_BAUD_56K	0x00008000	56 Kbps
HHD_BAUD_128K	0x00010000	128 Kbps
HHD_BAUD_115200	0x00020000	115200 bps
HHD_BAUD_57600	0x00040000	57600 bps
HHD_BAUD_USER	0x10000000	Programmable baud rate

DATABITS_ST Enumeration

Description

Data bits settable parameters (bit mask)

Symbol	Value	Description
HHD_DATABITS_5	0x00000001	5 data bits
HHD_DATABITS_6	0x00000002	6 data bits
HHD_DATABITS_7	0x00000004	7 data bits
HHD_DATABITS_8	0x00000008	8 data bits
HHD_DATABITS_16	0x00000010	16 data bits
HHD_DATABITS_16X	0x00000020	Special wide path through serial hardware lines

EVENTS Enumeration

Symbol	Value	Description
HHD_SERIAL_EV_RXCHAR	0x00000001	Any Character received
HHD_SERIAL_EV_RXFLAG	0x00000002	Received certain character
HHD_SERIAL_EV_TXEMPTY	0x00000004	Transmitt Queue Empty
HHD_SERIAL_EV_CTS	0x00000008	CTS changed state
HHD_SERIAL_EV_DSR	0x00000010	DSR changed state
HHD_SERIAL_EV_RLSD	0x00000020	RLSD changed state
HHD_SERIAL_EV_BREAK	0x00000040	BREAK received
HHD_SERIAL_EV_ERR	0x00000080	Line status error occurred
HHD_SERIAL_EV_RING	0x00000100	Ring signal detected
HHD_SERIAL_EV_PERR	0x00000200	Printer error occured
HHD_SERIAL_EV_RX80FULL	0x00000400	Receive buffer is 80 percent full
HHD_SERIAL_EV_EVENT1	0x00000800	Provider specific event 1
HHD_SERIAL_EV_EVENT2	0x00001000	Provider specific event 2

MODEMSTATUS Enumeration

Description

Modem status flags. These masks are used to access the modem status register. Whenever one of the first four bits in the modem status register changes state a modem status interrupt is generated.

Symbol	Value	Description
HHD_SERIAL_MSR_DCTS	0x00000001	This bit is the delta clear to send. It is used to indicate that the clear to send bit (in this register) has *changed* since this register was last read by the CPU.
HHD_SERIAL_MSR_DDSR	0x00000002	This bit is the delta data set ready. It is used to indicate that the data set ready bit (in this register) has *changed* since this register was last read by the CPU.
HHD_SERIAL_MSR_TERI	0x00000004	This is the trailing edge ring indicator. It is used to indicate that the ring indicator input has changed from a low to high state.
HHD_SERIAL_MSR_DDCD	0x00000008	This bit is the delta data carrier detect. It is used to indicate that the data carrier bit (in this register) has *changed* since this register was last read by the CPU.
HHD_SERIAL_MSR_CTS	0x00000010	This bit contains the (complemented) state of the clear to send (CTS) line.
HHD_SERIAL_MSR_DSR	0x00000020	This bit contains the (complemented) state of the data set ready (DSR) line.
HHD_SERIAL_MSR_RI	0x00000040	This bit contains the (complemented) state of the ring indicator (RI) line.
HHD_SERIAL_MSR_DCD	0x00000080	This bit contains the (complemented) state of the data carrier detect (DCD) line.

PARITY Enumeration

Symbol	Value	Description
HHD_NO_PARITY	0x00000000	No parity.
HHD_ODD_PARITY	0x00000001	Odd parity.
HHD_EVEN_PARITY	0x00000002	Even parity.
HHD_MARK_PARITY	0x00000003	Mark parity.
HHD_SPACE_PARITY	0x00000004	Space parity.

PROVIDERCAPS Enumeration

Description

Provider capabilities flags.

Symbol	Value	Description
HHD_PCF_DTRDSR	0x00000001	DTR (data-terminal-ready)/DSR (data-set-ready) supported
HHD_PCF_RTSCTS	0x00000002	RTS (request-to-send)/CTS (clear-to-send) supported
HHD_PCF_RLSD	0x00000004	RLSD (receive-line-signal-detect) supported
HHD_PCF_PARITY_CHECK	0x00000008	Parity checking supported
HHD_PCF_XONXOFF	0x00000010	XON/XOFF flow control supported
HHD_PCF_SETXCHAR	0x00000020	Settable XON/XOFF supported
HHD_PCF_TOTALTIMEOUTS	0x00000040	Total (elapsed) time-outs supported
HHD_PCF_INTTIMEOUTS	0x00000080	Interval time-outs supported
HHD_PCF_SPECIALCHARS	0x00000100	Special character support provided
HHD_PCF_16BITMODE	0x00000200	Special 16-bit mode supported

PROVIDERSETTABLE Enumeration

Description

Settable provider parameters.

Symbol	Value	Description
HHD_SP_PARITY	0x00000001	Parity
HHD_SP_BAUD	0x00000002	Baud rate
HHD_SP_DATABITS	0x00000004	Data bits
HHD_SP_STOPBITS	0x00000008	Stop bits
HHD_SP_HANDSHAKING	0x00000010	Handshaking (flow control)
HHD_SP_PARITY_CHECK	0x00000020	Parity checking
HHD_SP_RLSD	0x00000040	RLSD (receive-line-signal-detect)

PROVIDERTYPE Enumeration

Symbol	Value	Description
HHD_PST_UNSPECIFIED	0x00000000	Unspecified
HHD_PST_RS232	0x00000001	RS-232 serial port
HHD_PST_PARALLELPORT	0x00000002	Parallel port
HHD_PST_RS422	0x00000003	RS-422 port
HHD_PST_RS423	0x00000004	RS-423 port
HHD_PST_RS449	0x00000005	RS-449 port
HHD_PST_MODEM	0x00000006	Modem device
HHD_PST_FAX	0x00000021	FAX device
HHD_PST_SCANNER	0x00000022	Scanner device
HHD_PST_NETWORK_BRIDGE	0x00000100	Unspecified network bridge
HHD_PST_LAT	0x00000101	LAT protocol
HHD_PST_TCPIP_TELNET	0x00000102	TCP/IP telnet protocol
HHD_PST_X25	0x00000103	X.25 standards

PURGE Enumeration

Symbol	Value	Description
HHD_SERIAL_PURGE_TXABORT	0x00000001	Terminates all outstanding overlapped write operations and returns immediately, even if the write operations have not been completed.
HHD_SERIAL_PURGE_RXABORT	0x00000002	Terminates all outstanding overlapped read operations and returns immediately, even if the read operations have not been completed.
HHD_SERIAL_PURGE_TXCLEAR	0x00000004	Clears the output buffer (if the device driver has one).
HHD_SERIAL_PURGE_RXCLEAR	0x00000008	Clears the input buffer (if the device driver has one).

STOPBITS Enumeration

Symbol	Value	Description
HHD_STOP_BIT_1	0x00000000	1 stop bit
HHD_STOP_BITS_1_5	0x00000001	1.5 stop bits
HHD_STOP_BITS_2	0x00000002	2 stop bits

STOPPARITY_ST Enumeration

Symbol	Value	Description
`HHD_STOPBITS_10`	`0x00000001`	1 stop bit
`HHD_STOPBITS_15`	`0x00000002`	1.5 stop bits
`HHD_STOPBITS_20`	`0x00000004`	2 stop bits
`HHD_PARITY_NONE`	`0x00000100`	No parity
`HHD_PARITY_ODD`	`0x00000200`	Odd parity
`HHD_PARITY_EVEN`	`0x00000400`	Even parity
`HHD_PARITY_MARK`	`0x00000800`	Mark parity
`HHD_PARITY_SPACE`	`0x00001000`	Space parity

ConnectionState Enumeration

Symbol	Value	Description
`DeviceDisconnected`	`0x00000000`	Illustrates that the device the monitor attached to has been disconnected from the system.
`DeviceConnected`	`0x00000001`	Illustrates that the device the monitor has connected to the device.