PCAN-OBD-2 API

API Implementation of the OBD-2 Standard (ISO 15765-4)

Documentation
# Contents

1 PCAN-OBD-2 API 5

2 Introduction 6
   2.1 Understanding PCAN-OBD-2 6
   2.2 Using PCAN-OBD-2 6
   2.3 License Regulations 7
   2.4 Features 7
   2.5 System Requirements 8
   2.6 Scope of Supply 8

3 DLL API Reference 9
   3.1 Namespaces 9
      3.1.1 Peak.Can.ObdII 9
   3.2 Units 10
      3.2.1 POBDII Unit 10
   3.3 Classes 11
      3.3.1 OBDIIApi 12
      3.3.2 TObdiiApi 12
   3.4 Structures 13
      3.4.1 TOBDIIResponse 13
      3.4.2 TPOBDIIParamData 14
      3.4.3 TPOBDIIDTCText 16
      3.4.4 TPOBDIIDTCData 17
      3.4.5 TPOBDIIIMonitorData 18
      3.4.6 TPOBDIIIInfoData 20
      3.4.7 TPOBDIIUnitAndScaling 21
   3.5 Types 23
      3.5.1 TPOBDIICANHandle 23
      3.5.2 TPOBDIIPid 24
      3.5.3 TPOBDIOBDMid 25
      3.5.4 TPOBDIITid 25
      3.5.5 TPOBDIIIInfoType 26
      3.5.6 TPOBDIIIBaudrateInfo 27
      3.5.7 TPOBDIIAddress 28
      3.5.8 TPOBDIIIService 30
      3.5.9 TPOBDIIParameter 31
      3.5.10 TPOBDIIHWType 37
      3.5.11 TPOBDIISStatus 38
      3.5.12 TPOBDIIIError 41
      3.5.13 TPOBDIIIInfoDataType 42
   3.6 Methods 44
      3.6.1 Initialize 44
      3.6.2 Initialize (TPOBDIICANHandle) 45
      3.6.3 Initialize (TPOBDIICANHandle, TPOBDIIIBaudrateInfo, TPOBDIIHWType, UInt32, UInt16) 47
3.6.4 Uninitialize 50
3.6.5 SetValue 52
3.6.6 GetValue 55
3.6.7 GetValue (TPOBDIICANHandle, TPOBDIIParameter, StringBuilder, UInt32) 55
3.6.8 GetValue (TPOBDIICANHandle, TPOBDIIParameter, UInt32, UInt32) 57
3.6.9 GetValue (TPOBDIICANHandle, TPOBDIIParameter, Byte[], UInt32) 60
3.6.10 GetStatus 62
3.6.11 Reset 65
3.6.12 GetUnitAndScaling 67
3.6.13 RequestCurrentData 69
3.6.14 RequestFreezeFrameData 73
3.6.15 RequestStoredTroubleCodes 76
3.6.16 ClearTroubleCodes 80
3.6.17 RequestTestResults 83
3.6.18 RequestPendingTroubleCodes 87
3.6.19 RequestControlOperation 91
3.6.20 RequestVehicleInformation 94
3.6.21 RequestPermanentTroubleCodes 98

3.7 Functions 102
3.7.1 OBDII_Initialize 103
3.7.2 OBDII_Uninitialize 104
3.7.3 OBDII_SetValue 105
3.7.4 OBDII_GetValue 106
3.7.5 OBDII_GetStatus 107
3.7.6 OBDII_Reset 108
3.7.7 OBDII_GetUnitAndScaling 109
3.7.8 OBDII_RequestCurrentData 110
3.7.9 OBDII_RequestFreezeFrameData 111
3.7.10 OBDII_RequestStoredTroubleCodes 113
3.7.11 OBDII_ClearTroubleCodes 114
3.7.12 OBDII_RequestTestResults 115
3.7.13 OBDII_RequestPendingTroubleCodes 117
3.7.14 OBDII_RequestControlOperation 118
3.7.15 OBDII_RequestVehicleInformation 119
3.7.16 OBDII_RequestPermanentTroubleCodes 121

3.8 Definitions 123
3.8.1 PCAN-OBD-2 Handle Definitions 123
3.8.2 Parameter Value Definitions 124

4 Additional Information 126
4.1 PCAN Fundamentals 126
4.2 PCAN-Basic 127
4.3 OBDII, UDS, and ISO-TP Network Addressing Information 129
   4.3.1 UDS and ISO-TP Network Addressing Information 129
   4.3.2 ISO-TP Network Addressing Format 129
1 PCAN-OBD-2 API

Welcome to the documentation of PCAN-OBD-2 API, a PEAK CAN API that implements ISO 15765-4, Diagnostics on CAN - Requirements for emissions-related system, an international standard that specifies the type of diagnostic connector and its pinout, the electrical signaling protocols available, the messaging format and a candidate list of vehicle parameters to monitor (along with information how to encode the data for each).

In the following chapters you will find all the information needed to take advantage of this API.

- Introduction on page 6
- DLL API Reference on page 9
- Additional Information on page 126
2 Introduction

PCAN-OBD-2 is a simple programming interface intended to support windows automotive applications that use PEAK-Hardware to communicate with Electronic Control Units (ECU) connected to the bus systems of a car, for maintenance purpose.

2.1 Understanding PCAN-OBD-2

OBD-II stands for On-Board Diagnostics. It is a standard that specifies the type of diagnostic connector and its pinout, the electrical signaling protocols available, the messaging format and a candidate list of vehicle parameters to monitor along with information how to encode the data for each.

The OBD communication protocol is defined in SAE J1979 / ISO 15031-5 and the specific implementation for CAN bus is described in ISO 15765-4. It is a Client/Server oriented protocol:

- External test equipment has the role of the client,
- Electronic Control Units (ECUs) connected inside the vehicle are servers.

Note: There are five signaling protocols that are permitted with the OBD-II interface. PCAN-OBD-2 is only valid with ISO 15765 CAN signals. PCAN-OBD-2 is the collection of the following standards: ISO 15765-4, ISO 15765-3 (known as Diagnostic/UDS on CAN), and ISO 15765-2 (known as ISO-TP).

Client always starts the communication by sending a request. If a server supports this request, it will reply with a positive or negative response, otherwise it can just ignore request. As the client cannot choose which server should reply to his request, and that request can be supported by multiple servers, client should be aware of receiving more than one response to a request.

Since the transport protocol of OBD-II on CAN is done using ISO-TP, an international standard for sending data packets over a CAN Bus, the maximum data length that can be transmitted in a single data-block is 4095 bytes.

PCAN-OBD-2 API is an implementation of the OBD-II on CAN standard. The physical communication is carried out by PCAN-Hardware (PCAN-USB, PCAN-PCI etc.) through the PCAN-UDS, PCAN-ISO-TP and PCAN-Basic API (free CAN APIs from PEAK-System). Because of this it is necessary to have also the PCAN-UDS, PCAN-ISO-TP and PCAN-Basic APIs (PCAN-UDS.dll, PCAN-ISO-TP.dll and PCANBasic.dll) present on the working computer where PCAN-OBD-2 is intended to be used. PCAN-OBD-2, PCAN-UDS, PCAN-ISO-TP and PCAN-Basic APIs are free and available for all people that acquire a PCAN-Hardware.

2.2 Using PCAN-OBD-2

Since PCAN-OBD-2 API is built on top of the PCAN-UDS API, PCAN-ISO-TP API and PCAN-Basic APIs, it shares similar functions. It offers the possibility to use several PCAN-OBD-2 (POBDII) Channels within the same application in an easy way. The communication process is divided in 3 phases: initialization, interaction and finalization of a POBDII-Channel.
**Initialization:** In order to do OBD-II on CAN communication using a channel, it is necessary to initialize it first. This is done by making a call to the function `OBDII.Initialize` (**class-method**: `Initialize`).

**Interaction:** After a successful initialization, a channel is ready to communicate with the connected CAN bus. Further configuration is not needed. The 9 functions starting with `OBDII_Request` (**class-methods**: starting with `Request`) can be used to transmit legislated-OBD requests and receive the responses of the ECUs.

**Finalization:** When the communication is finished, the function `OBDII.Uninitialize` (**class-method**: `Uninitialize`) should be called in order to release the POBDII-Channel and the resources allocated for it. In this way the channel is marked as "Free" and can be used from other applications.

### 2.3 License Regulations

The interface DLLs of this API, PCAN-Basic, device drivers, and further files needed for linking are property of the PEAK-System Technik GmbH and may be used only in connection with a hardware component purchased from PEAK-System or one of its partners. If a CAN hardware component of third-party suppliers should be compatible to one of PEAK-System, then you are not allowed to use or to pass on the APIs and driver software of PEAK-System.

If a third-party supplier develops software based on the PCAN-OBD-II API and problems occur during the use of this software, consult the software provider.

### 2.4 Features

- Implementation of the OBD-2 protocol (ISO 15765-4) as on-board diagnostics standard
- Windows DLLs for the development of applications for the platforms Windows® 11 (x64/ARM64), 10 (x64/x86)
- Thread-safe API
- Physical communication via CAN using a CAN interface of the PCAN series
- Uses the PCAN-Basic programming interface to access the CAN hardware in the computer
- Uses the PCAN-ISO-TP programming interface (ISO 15765-2) for the transfer of data packages up to 4095 bytes via the CAN bus
- Uses the PCAN-UDS programming interface (ISO 14229-1) for the communication with control units
2.5 System Requirements

- Windows 11 (x64/ARM64), Windows 10 (x64)
- For the CAN bus connection: PC CAN interface from PEAK-System
- PCAN-Basic API
- PCAN-ISO-TP API
- PCAN-UDS API

2.6 Scope of Supply

- Interface DLLs for Windows (x86/x64/ARM64)
- Examples and header files for all common programming languages
- Documentation in PDF format
3 DLL API Reference

This section contains information about the data types (classes, structures, types, defines, enumerations) and API functions which are contained in the PCAN-OBD-2 API.

3.1 Namespaces

**PEAK** offers the implementation of some specific programming interfaces as namespaces for the .NET Framework programming environment. The following namespaces are available:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak</td>
<td>Contains all namespaces that are part of the managed programming environment from PEAK-System</td>
</tr>
<tr>
<td>Peak.Can</td>
<td>Contains types and classes for using the PCAN API from PEAK-System</td>
</tr>
<tr>
<td>Peak.Can.Light</td>
<td>Contains types and classes for using the PCAN-Light API from PEAK-System</td>
</tr>
<tr>
<td>Peak.Can.Basic</td>
<td>Contains types and classes for using the PCAN-Basic API from PEAK-System</td>
</tr>
<tr>
<td>Peak.Can.Ccp</td>
<td>Contains types and classes for using the CCP API implementation from PEAK-System</td>
</tr>
<tr>
<td>Peak.Can.Xcp</td>
<td>Contains types and classes for using the XCP API implementation from PEAK-System</td>
</tr>
<tr>
<td>Peak.Can.Iso.Tp</td>
<td>Contains types and classes for using the PCAN-ISO-TP API implementation from PEAK-System</td>
</tr>
<tr>
<td>Peak.Can.Uds</td>
<td>Contains types and classes for using the PCAN-UDS API implementation from PEAK-System</td>
</tr>
<tr>
<td>Peak.Can.ObdII</td>
<td>Contains types and classes for using the PCAN-OBD-2 API implementation from PEAK-System</td>
</tr>
<tr>
<td>Peak.Lin</td>
<td>Contains types and classes used to handle with LIN devices from PEAK-System</td>
</tr>
<tr>
<td>Peak.RP1210A</td>
<td>Contains types and classes used to handle with CAN devices from PEAK-System through the TMC Recommended Practices 1210, version A, as known as RP1210(A)</td>
</tr>
</tbody>
</table>

3.1.1 Peak.Can.ObdII

The Peak.Can.ObdII namespace contains types and classes to use the PCAN-OBD-2 API within the .NET Framework programming environment and handle PCAN devices from PEAK-System.

**Remarks:** Under the Delphi environment, these elements are enclosed in the POBDII-Unit. The functionality of all elements included here is just the same. The difference between this namespace and the Delphi unit consists in the fact that Delphi accesses the Windows API directly (it is not Managed Code).

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPOBDIIHandle</td>
<td>Represents a PCAN-OBD-2 channel handle</td>
</tr>
<tr>
<td>TPOBDIIPid</td>
<td>Represents a Parameter Identification (PID), a parameter used by OBD-II service $01 and $02</td>
</tr>
<tr>
<td>TPOBDIIOBDMid</td>
<td>Represents an On-Board Diagnostic Monitor ID (OBDMID), a parameter used by OBD-II service $06</td>
</tr>
<tr>
<td>TPOBDIITid</td>
<td>Represents a Test ID (TID), a parameter used by OBD-II service $08</td>
</tr>
<tr>
<td>TPOBDIIInfoType</td>
<td>Represents an InfoType, a parameter used by OBD-II service $09</td>
</tr>
</tbody>
</table>
Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBDIIApi</td>
<td>Defines a class which represents the PCAN-OBD-2 API</td>
</tr>
</tbody>
</table>

Structures

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPOBDIIResponse</td>
<td>Represents the generic data returned by OBD services</td>
</tr>
<tr>
<td>TPOBDIIParamData</td>
<td>Represents the data returned by OBD service $01, $02</td>
</tr>
<tr>
<td>TPOBDIIIDTCText</td>
<td>Represents a DTC as a string of 6 bytes</td>
</tr>
<tr>
<td>TPOBDIIIDTCData</td>
<td>Represents the data returned by OBD service $03, $07, $0A</td>
</tr>
<tr>
<td>TPOBDIIIMonitorData</td>
<td>Represents the data returned by OBD service $06</td>
</tr>
<tr>
<td>TPOBDIIInfoData</td>
<td>Represents the data returned by OBD service $09</td>
</tr>
<tr>
<td>TPOBDIIUnitAndScaling</td>
<td>Represents a Unit and Scaling definition</td>
</tr>
</tbody>
</table>

Enumerations

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPOBDIIBaudrateInfo</td>
<td>Represents a legislated OBD-II Baud rate</td>
</tr>
<tr>
<td>TPOBDIIAddress</td>
<td>Represents a legislated OBD-II address for ECUs</td>
</tr>
<tr>
<td>TPOBDIIService</td>
<td>Represents an OBD-II Service</td>
</tr>
<tr>
<td>TPOBDIIParameter</td>
<td>Represents a PCAN-OBD-2 parameter to be read or set</td>
</tr>
<tr>
<td>TPOBDIIHwType</td>
<td>Represents the type of CAN hardware to be initialized</td>
</tr>
<tr>
<td>TPOBDIIStatus</td>
<td>Represents an OBD-II status/error code</td>
</tr>
<tr>
<td>TPOBDIIError</td>
<td>Represents an OBD-II Response codes</td>
</tr>
<tr>
<td>TPOBDIIInfoDataType</td>
<td>Represents the type of returned data in TPOBDIIInfoData (OBD service $09)</td>
</tr>
</tbody>
</table>

3.2 Units

PEAK offers the implementation of some specific programming interfaces as Units for the Delphi’s programming environment. The following units are available to be used:

Namespaces

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>POBDII Unit</td>
</tr>
<tr>
<td></td>
<td>Delphi unit for using the PCAN-OBD-2 API from PEAK-System</td>
</tr>
</tbody>
</table>

3.2.1 POBDII Unit

The POBDII-Unit contains types and classes to use the PCAN-OBD-2 API within Delphi’s programming environment and handle PCAN devices from PEAK-System.

Remarks: For the .NET Framework, these elements are enclosed in the Peak.Can.Obdii namespace. The functionality of all elements included here is just the same. The difference between this Unit and the .NET namespace consists in the fact that Delphi accesses the Windows API directly (it is not Managed Code).
### Aliases

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPOBDIIICANHandle</td>
<td>Represents a PCAN-OBD-2 channel handle</td>
</tr>
<tr>
<td>TPOBDIIPid</td>
<td>Represents a Parameter Identification (PID), a parameter used by OBD-II service $01 and $02</td>
</tr>
<tr>
<td>TPOBDIIIOBDMid</td>
<td>Represents an On-Board Diagnostic Monitor ID (OBDMID), a parameter used by OBD-II service $06</td>
</tr>
<tr>
<td>TPOBDIITid</td>
<td>Represents a Test ID (TID), a parameter used by OBD-II service $08</td>
</tr>
<tr>
<td>TPOBDIIInfoType</td>
<td>Represents an InfoType, a parameter used by OBD-II service $09</td>
</tr>
</tbody>
</table>

### Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBDIIApi</td>
<td>Defines a class which represents the PCAN-OBD-2 API</td>
</tr>
</tbody>
</table>

### Structures

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPOBDIIResponse</td>
<td>Represents the generic data returned by OBD services</td>
</tr>
<tr>
<td>TPOBDIIParamData</td>
<td>Represents the data returned by OBD service $01, $02</td>
</tr>
<tr>
<td>TPOBDIITCText</td>
<td>Represents a DTC as a string of 6 bytes</td>
</tr>
<tr>
<td>TPOBDIIDTCData</td>
<td>Represents the data returned by OBD service $03, $07, $0A</td>
</tr>
<tr>
<td>TPOBDIIMonitorData</td>
<td>Represents the data returned by OBD service $06</td>
</tr>
<tr>
<td>TPOBDIIInfoData</td>
<td>Represents the data returned by OBD service $09</td>
</tr>
<tr>
<td>TPOBDIIUnitAndScaling</td>
<td>Represents a Unit and Scaling definition</td>
</tr>
</tbody>
</table>

### Enumerations

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPOBDIIIBaudrateInfo</td>
<td>Represents a legislated OBD-II Baud rate</td>
</tr>
<tr>
<td>TPOBDIIService</td>
<td>Represents a legislated OBD-II address for ECUs</td>
</tr>
<tr>
<td>TPOBDIIService</td>
<td>Represents an OBD-II Service</td>
</tr>
<tr>
<td>TPOBDIIParameter</td>
<td>Represents a PCAN-OBD-2 parameter to be read or set</td>
</tr>
<tr>
<td>TPOBDIIHwType</td>
<td>Represents the type of CAN hardware to be initialized</td>
</tr>
<tr>
<td>TPOBDIIStatus</td>
<td>Represents an OBD-II status/error code</td>
</tr>
<tr>
<td>TPOBDIIError</td>
<td>Represents an OBD-II Response codes</td>
</tr>
<tr>
<td>TPOBDIIInfoDataType</td>
<td>Represents the type of returned data in TPOBDIIInfoData (OBD service $09)</td>
</tr>
</tbody>
</table>

### 3.3 Classes

The following classes are offered to make use of the PCAN-OBD-2 API in a managed or unmanaged way.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBDII Api</td>
<td>Defines a class to use the PCAN-OBD-2 API within the Microsoft’s .NET Framework programming environment</td>
</tr>
<tr>
<td>TObdiiApi</td>
<td>Defines a class to use the PCAN-OBD-2 API within the Delphi programming environment</td>
</tr>
</tbody>
</table>
3.3.1 OBDIIApi
Defines a class which represents the PCAN-OBD-2 API to be used within the Microsoft's .NET Framework.

Syntax
C#
public static class OBDIIApi

C++ / CLR
public ref class OBDIIApi abstract sealed

Visual Basic
Public NotInheritable Class OBDIIApi

Remarks: The OBDIIApi class collects and implements the PCAN-OBD-2 API functions. Each method is called just like the API function with the exception that the prefix "OBDII_" is not used. The structure and functionality of the methods and API functions are the same.

Within the .NET Framework from Microsoft, the OBDIIApi class is a static, not inheritable, class. It can (must) directly be used, without any instance of it, e.g.:

```
TPOBDIIStatus res;
// Static use, without any instance
// res = OBDIIApi.Initialize(OBDIIApi.POBDII_USBBUS1);
```

Note: This class under Delphi is called TObdiiApi.

See also: Methods on page 44, Definitions on page 123.

3.3.2 TObdiiApi
Defines a class which represents the PCAN-OBD-2 API to be used within the Delphi programming environment.

Syntax
Pascal OO
TObdiiApi = class

Remarks: TObdiiApi is a class containing only class-methods and constant members, allowing their use without the creation of any object, just like a static class of another programming language. It collects and implements the PCAN-OBD-2 API functions. Each method is called just like the API function with the exception that the prefix "OBDII_" is not used. The structure and functionality of the methods and API functions are the same.

Note: This class under .NET framework is called OBDIIApi.

See also: Methods on page 44, Definitions on page 123.
3.4 Structures

The PCAN-OBD-2 API defines the following structures:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPOBDIIResponse</td>
<td>Represents the generic data returned by OBD services</td>
</tr>
<tr>
<td>TPOBDIIParamData</td>
<td>Represents the data returned by OBD service $01, $02</td>
</tr>
<tr>
<td>TPOBDIIDTCText</td>
<td>Represents a DTC as a string of 6 bytes</td>
</tr>
<tr>
<td>TPOBDIIDTCData</td>
<td>Represents the data returned by OBD service $03, $07, $0A</td>
</tr>
<tr>
<td>TPOBDIMonitorData</td>
<td>Represents the data returned by OBD service $06</td>
</tr>
<tr>
<td>TPOBDIIInfoData</td>
<td>Represents the data returned by OBD service $09</td>
</tr>
<tr>
<td>TPOBDIIUnitAndScaling</td>
<td>Represents a Unit and Scaling definition</td>
</tr>
</tbody>
</table>

3.4.1 TPOBDIIResponse

Defines a generic response structure to any OBD services. Such structure contains the raw data.

Syntax

C++

typedef struct tagTPOBDIIResponse
{
    BYTE   SOURCE;
    TPOBDIIError ERRORNR;
    BYTE   DATA[4095];
    WORD   LEN;
    BYTE   SID;
    BYTE   PID;
    BYTE   FRAME;
} TPOBDIIResponse;

Pascal OO

TPOBDIIResponse = record
    SOURCE: Byte;
    ERRORNR: TPOBDIIError;
    DATA: array[0..4094] of Byte;
    LEN: WORD;
    SID: Byte;
    PID: Byte;
    FRAME: Byte;
end;
PTPOBDIIResponse = ^TPOBDIIResponse;

C#

[StructLayout(LayoutKind.Sequential)]
public struct TPOBDIIResponse
{
    public byte SOURCE;
    [MarshalAs(UnmanagedType.U1)]
    public TPOBDIIError ERRORNR;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 4095)]
    public byte[] DATA;
    public ushort LEN;
public byte SID;
public byte PID;
public byte FRAME;
}

### C++ / CLR

```csharp
[StructLayout(LayoutKind::Sequential)]
public value struct TPOBDIIResponse
{
    Byte SOURCE;
    [MarshalAs(UnmanagedType::U1)]
    TPOBDIIError ERRORNR;
    [MarshalAs(UnmanagedType::ByValArray, SizeConst = 4095)]
    array<Byte>^ DATA;
    unsigned short LEN;
    Byte SID;
    Byte PID;
    Byte FRAME;
};
```

### Visual Basic

```vbnet
<StructLayout(LayoutKind.Sequential)> _
Public Structure TPOBDIIResponse
    Public SOURCE As Byte
    <MarshalAs(UnmanagedType::U1)> _
    Public ERRORNR As TPOBDIIError
    <MarshalAs(UnmanagedType::ByValArray, SizeConst:=4095)> _
    Public DATA As Byte()
    Public LEN As UShort
    Public SID As Byte
    Public PID As Byte
    Public FRAME As Byte
End Structure
```

### Fields

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOURCE</td>
<td>Source address of the response (usually the ECU number).</td>
</tr>
<tr>
<td>ERRORNR</td>
<td>Network error code (see TPOBDIIError).</td>
</tr>
<tr>
<td>DATA</td>
<td>Buffer containing the raw data of the response, note that SID and parameters have been removed.</td>
</tr>
<tr>
<td>LEN</td>
<td>Number of bytes stored in the DATA buffer.</td>
</tr>
<tr>
<td>SID</td>
<td>Requested Service ID.</td>
</tr>
<tr>
<td>PID</td>
<td>Requested Parameter ID if present (only with OBD services $01, $02, $08 and $09).</td>
</tr>
<tr>
<td>FRAME</td>
<td>Requested frame number if present (only with OBD service $02).</td>
</tr>
</tbody>
</table>

See also: `TPOBDIIError` on page 41, `TPOBDIIAddress` on page 28.

### 3.4.2 TPOBDIIParamData

Represents the responses returned by OBD services $01 and $02: respectively `RequestCurrentData` and `RequestFreezeFrameData`.

### Syntax
C++

typedef struct tagTPOBDIIParamData
{
    TPOBDIIResponse RESPONSE;
    BYTE BUFFER[41];
    double DOUBLES[10];
    WORD BYTEMASK;
    BYTE BLEN;
    BYTE DLEN;
    char DTC[6];
} TPOBDIIParamData;

Pascal OO

TPOBDIIParamData = record
    RESPONSE: TPOBDIIResponse;
    BUFFER: array[0..40] of Byte;
    DOUBLES: array[0..9] of Double;
    BYTEMASK: Word;
    BLEN: Byte;
    DLEN: Byte;
    DTC: array[0..5] of Char;
end;
PTPOBDIIParamData = ^TPOBDIIParamData;

C#

public struct TPOBDIIParamData
{
    [MarshalAs(UnmanagedType.Struct)]
    public TPOBDIIResponse RESPONSE;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 41)]
    public byte[] BUFFER;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 10)]
    public double[] DOUBLES;
    public ushort BYTEMASK;
    public byte BLEN;
    public byte DLEN;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 6)]
    public string DTC;
}

C++ / CLR

public value struct TPOBDIIParamData
{
    TPOBDIIResponse RESPONSE;
    [MarshalAs(UnmanagedType::ByValArray, SizeConst = 41)]
    array<Byte>^ BUFFER;
    [MarshalAs(UnmanagedType::ByValArray, SizeConst = 10)]
    array<double>^ DOUBLES;
    unsigned short BYTEMASK;
    Byte BLEN;
    Byte DLEN;
    [MarshalAs(UnmanagedType::ByValTStr, SizeConst = 6)]
    String^ DTC;
};
Public Structure TPOBDIIParamData
    <MarshalAs(UnmanagedType.UnmanagedType.Struct)> _
    Public RESPONSE As TPOBDIIResponse
    <MarshalAs(UnmanagedType.UnmanagedType.ByValArray, SizeConst:=41)> _
    Public BUFFER As Byte()
    <MarshalAs(UnmanagedType.UnmanagedType.ByValArray, SizeConst:=10)> _
    Public DOUBLES As Double()
    Public BYTEMASK As UShort
    Public BLEN As Byte
    Public DLEN As Byte
    <MarshalAs(UnmanagedType.UnmanagedType.ByValTStr, SizeConst:=6)> _
    Public DTC As String
End Structure

Fields

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESPONSE</td>
<td>The generic raw response</td>
</tr>
<tr>
<td>BUFFER</td>
<td>Data as an array of Bytes (up to 41 values)</td>
</tr>
<tr>
<td>DOUBLES</td>
<td>Data as an array of Doubles (up to 10 values)</td>
</tr>
<tr>
<td>BYTEMASK</td>
<td>Mask for bytes that are bit encoded (depending on the PID requested)</td>
</tr>
<tr>
<td>BLEN</td>
<td>Number of bytes stored in buffer BUFFER (0 to 41)</td>
</tr>
<tr>
<td>DLEN</td>
<td>Number of doubles stored in buffer DOUBLES (0 to 41)</td>
</tr>
<tr>
<td>DTC</td>
<td>A single DTC (as a string including the ‘\0’ character)</td>
</tr>
</tbody>
</table>

Remarks: Depending on the request, buffers (BUFFER or DOUBLES) may have data or not; check the BLEN, DLEN, and DTC parameters to see if the size is greater than zero.

Note: Non-standard (i.e. vehicle-specific) data are not handle and will not be parsed successfully, in those cases user will have to check the RESPONSE field to get the raw data.

See also: TOBDIIResponse on page 13, OBDII_RequestCurrentData on page 110 (class-method: RequestCurrentData), OBDII_RequestFreezeFrameData on page 111 (class-method: RequestFreezeFrameData)

3.4.3 TPOBDIIDTCText
Defines a string to store a Diagnostic Trouble Code (DTC).

Pascal OO

TPOBDIIDTCText = record
    ErrorText: String;
end;

C#

public struct TPOBDIIDTCText
{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 6)]
    public string ErrorText;
}
C++ / CLR

```c++
public value struct TPOBDIIDTCText
{
    [MarshalAs(UnmanagedType::ByValTStr, SizeConst = 6)]
    String^ ErrorText;
};
```

Visual Basic

```vbnet
Public Structure TPOBDIIDTCText
    <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=6)> Public ErrorText As String
End Structure
```

### Fields

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrorText</td>
<td>The DTC (Diagnostic Trouble Code) as a string. Default length of DTC is 5</td>
</tr>
</tbody>
</table>

**Remarks:** In plain C++, this structure is not used and is replaced by an array of char (see TPOBDIIDTCData). See also: TPOBDIIDTCData below.

### 3.4.4 TPOBDIIDTCData

Represents the responses returned by OBD services $03$ and $07$ and $0A$: respectively RequestStoredTroubleCodes, RequestPendingTroubleCodes, and RequestPermanentTroubleCodes.

**Syntax**

**C++**

```c++
typedef struct tagTPOBDIIDTCData
{
    TPOBDIIResponse RESPONSE;
    char   DTC[10][6];
    BYTE   DLEN;
} TPOBDIIDTCData;
```

**Pascal OO**

```pascal
TPOBDIIDTCData = record
    RESPONSE: TPOBDIIResponse;
    DTC: array[0..9] of array[0..5] of Char;
    DLEN: Byte;
end;
PTPOBDIIDTCData = ^TPOBDIIDTCData;
```

**C#**

```csharp
public struct TPOBDIIDTCData
{
    [MarshalAs(UnmanagedType.Struct)]
    public TPOBDIIResponse RESPONSE;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 10)]
    public TPOBDIIDTCText[] DTC;
    public byte DLEN;
}
```
C++ / CLR

```c++
public value struct TPOBDIIDTCDATA
{
    TPOBDIIResponse RESPONSE;
    [MarshalAs(UnmanagedType::ByValArray, SizeConst = 10)]
    array<TPOBDIIDTCText>^ DTC;
    Byte DLEN;
};
```

Visual Basic

```vbnet
Public Structure TPOBDIIDTCDATA
    <MarshalAs(UnmanagedType.Struct)> Public RESPONSE As TPOBDIIResponse
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=10)> Public DTC As TPOBDIIDTCText()
    Public DLEN As Byte
End Structure
```

### Fields

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESPONSE</td>
<td>The generic raw response</td>
</tr>
<tr>
<td>DTC</td>
<td>Buffer to store an array of TPOBDIIDTCText (note: in plain C++, DTC field is a two-dimensional array)</td>
</tr>
<tr>
<td>DLEN</td>
<td>Number of DTCs stored in the buffer</td>
</tr>
</tbody>
</table>

**Note:** Non-standard (i.e. vehicle-specific) data are not handle and will not be parsed successfully, in those cases user will have to check the RESPONSE field to get the raw data.

**See also:** [TPOBDIIDTCText on page 16](#), [OBDII_RequestStoredTroubleCodes on page 113](#) (class-method: RequestStoredTroubleCodes), [OBDII_RequestPendingTroubleCodes on page 117](#) (class-method: RequestPendingTroubleCodes), [OBDII_RequestPermanentTroubleCodes on page 121](#) (class-method: RequestPermanentTroubleCodes)

### 3.4.5 TPOBDIIMonitorData

Represents the responses returned by OBD service $06: RequestTestResults.

**Syntax**

**C++**

```c++
typedef struct tagTPOBDIIMonitorData
{
    TPOBDIIResponse RESPONSE;
    BYTE TID;
    BYTE UNITANDSCALING;
    double TESTVALUE;
    double MINLIMIT;
    double MAXLIMIT;
} TPOBDIIMonitorData;
```

**Pascal OO**

```pascal
TPOBDIIMonitorData = record
    RESPONSE: TPOBDIIResponse;
    TID: Byte;
```
UNITANDSCALING: Byte;
TESTVALUE: Double;
MINLIMIT: Double;
MAXLIMIT: Double;
end;
PTPOBDIIIMonitorData = ^TPOBDIIIMonitorData;

**C#**

```csharp
public struct TPOBDIIIMonitorData {
    [MarshalAs(UnmanagedType.Struct)]
    public TPOBDIIResponse RESPONSE;
    public byte TID;
    public byte UNITANDSCALING;
    public double TESTVALUE;
    public double MINLIMIT;
    public double MAXLIMIT;
}
```

**C++ / CLR**

```c++
public value struct TPOBDIIIMonitorData {
    TPOBDIIResponse RESPONSE;
    Byte TID;
    Byte UNITANDSCALING;
    double TESTVALUE;
    double MINLIMIT;
    double MAXLIMIT;
};
```

**Visual Basic**

```vbnet
Public Structure TPOBDIIIMonitorData
   <MarshalAs(UnmanagedType.Struct)> _
   Public RESPONSE As TPOBDIIResponse
   Public TID As Byte
   Public UNITANDSCALING As Byte
   Public TESTVALUE As Double
   Public MINLIMIT As Double
   Public MAXLIMIT As Double
End Structure
```

**Fields**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESPONSE</td>
<td>The generic raw response</td>
</tr>
<tr>
<td>TID</td>
<td>Test Identifier</td>
</tr>
<tr>
<td>UNITANDSCALING</td>
<td>Unit and Scaling Identifier (see GetUnitAndScaling or SAE J1979 Appendix E for definitions)</td>
</tr>
<tr>
<td>TESTVALUE</td>
<td>Test value (result, calculated based on the Unit and Scaling Id)</td>
</tr>
<tr>
<td>MINLIMIT</td>
<td>Minim Test Limit (calculated based on the Unit and Scaling Id)</td>
</tr>
<tr>
<td>MAXLIMIT</td>
<td>Maximum Test Limit (calculated based on the Unit and Scaling Id)</td>
</tr>
</tbody>
</table>

**Remarks:** Use the `GetUnitAndScaling` method to retrieve Unit and Scaling information based on an identifier.
Note: Non-standard (i.e. vehicle-specific) data are not handled and will not be parsed successfully, in those cases user will have to check the RESPONSE field to get the raw data.

See also: OBDII_GetUnitAndScaling on page 109 (class-method: GetUnitAndScaling), OBDII_RequestTestResults on page 115 (class-method: RequestTestResults)

3.4.6 TPOBDIIInfoData

Represents the responses returned by OBD service $09: RequestVehicleInformation.

Syntax

C++

typedef struct tagTPOBDIIInfoData
{
    TPOBDIIResponse RESPONSE;
    BYTE INDEXNR;
    TPOBDIIInfoDataType DATATYPE;
    WORD COUNTER;
    BYTE CALDATA[4];
    char TEXT[21];
} TPOBDIIInfoData;

Pascal OO

TPOBDIIInfoData = record
    RESPONSE: TPOBDIIResponse;
    INDEXNR: Byte;
    DATATYPE: TPOBDIIInfoDataType;
    COUNTER: Word;
    CALDATA: array[0..3] of Byte;
    TEXT: array[0..20] of char;
end;
PTPOBDIIInfoData = ^TPOBDIIInfoData;

C#

public struct TPOBDIIInfoData
{
    [MarshalAs(UnmanagedType.Struct)]
    public TPOBDIIResponse RESPONSE;
    public byte INDEXNR;
    [MarshalAs(UnmanagedType.U1)]
    public TPOBDIIInfoDataType DATATYPE;
    public ushort COUNTER;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 4)]
    public byte[] CALDATA;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 21)]
    public string TEXT;
}

C++ / CLR

[System::Runtime::InteropServices::StructLayout(LayoutKind::Sequential)]
public value struct TPOBDIIInfoData
{
    TPOBDIIResponse RESPONSE;
}
Visual Basic

```vbnet
Public Structure TPOBDIIInfoData
    <MarshalAs(UnmanagedType.Struct)> _
    Public RESPONSE As TPOBDIIResponse
    Public INDEXNR As Byte
        <MarshalAs(UnmanagedType.U1)> _
        Public DATATYPE As TPOBDIIInfoDataType
    Public COUNTER As UShort
        <MarshalAs(UnmanagedType.ByValArray, SizeConst:=4)> _
        Public CALDATA As Byte()
        <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=21)> _
        Public TEXT As String
End Structure
```

### Fields

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESPONSE</td>
<td>The generic raw response</td>
</tr>
<tr>
<td>INDEXNR</td>
<td>Index number of the data (depending on the type of the data)</td>
</tr>
<tr>
<td>DATATYPE</td>
<td>Type of the data (see TPOBDIIInfoDataType)</td>
</tr>
<tr>
<td>COUNTER</td>
<td>Data as a Counter value</td>
</tr>
<tr>
<td>CALDATA</td>
<td>Data as Calibration data</td>
</tr>
<tr>
<td>TEXT</td>
<td>Data as a String value</td>
</tr>
</tbody>
</table>

**Remarks:** TPOBDIIInfoData can store different data types, make sure to check the DATATYPE field to read the correct value.

**Note:** Non-standard (i.e. vehicle-specific) data are not handle and will not be parsed successfully, in those cases user will have to check the RESPONSE field to get the raw data.

**See also:** TPOBDIIInfoDataType on page 42

OBDII_RequestVehicleInformation on page 119 ([class-method]: RequestVehicleInformation)

### 3.4.7 TPOBDIIUnitAndScaling

Defines a Unit and Scaling structure ad defined in SAE J1979 Appendix E.

#### Syntax

**C++**

```cpp
typedef struct tagTPOBDIIUnitAndScaling
{
    double MIN;
    double MAX;
    CHAR UNIT[16];
} TPOBDIIUnitAndScaling;
```
Pascal OO

TPOBDIIUnitAndScaling = record
  MIN: Double;
  MAX: Double;
  UNIT_TXT: array[0..15] of Char;
end;
PTPOBDIIUnitAndScaling = ^TPOBDIIUnitAndScaling;

C#

public struct TPOBDIIUnitAndScaling
{
  public double MIN;
  public double MAX;
  [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 16)]
  public String UNIT;
};

C++ / CLR

public value struct TPOBDIIUnitAndScaling
{
  double MIN;
  double MAX;
  [MarshalAs(UnmanagedType::ByValTStr, SizeConst = 16)]
  String^ UNIT;
};

Visual Basic

Public Structure TPOBDIIUnitAndScaling
  Public MIN As Double
  Public MAX As Double
  <MarshalAs(UnmanagedType::ByValTStr, SizeConst:=16)> _
  Public UNIT As String
End Structure

Fields

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN</td>
<td>Minimum value for that unit</td>
</tr>
<tr>
<td>MAX</td>
<td>Maximum value for that unit</td>
</tr>
<tr>
<td>UNIT</td>
<td>Unit abbreviation as a string. Note that in Delphi as UNIT is a reserved keyword, it is replaced with UNIT_TXT</td>
</tr>
</tbody>
</table>

See also: OBDII_GetUnitAndScaling on page 109 (class-method: GetUnitAndScaling)
3.5 Types

The PCAN-ISO-TP API defines the following types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPOBDIICANHandle</td>
<td>Represents a PCAN-OBD-2 channel handle</td>
</tr>
<tr>
<td>TPOBDIPid</td>
<td>Represents a Parameter Identification (PID), a parameter used by OBD-II service $01 and $02</td>
</tr>
<tr>
<td>TPOBDIIOBDMid</td>
<td>Represents an On-Board Diagnostic Monitor ID (OBDMID), a parameter used by OBD-II service $06</td>
</tr>
<tr>
<td>TPOBDITid</td>
<td>Represents a Test ID (TID), a parameter used by OBD-II service $08</td>
</tr>
<tr>
<td>TPOBDIInfoType</td>
<td>Represents an InfoType, a parameter used by OBD-II service $09</td>
</tr>
<tr>
<td>TPOBDIIBaudrateInfo</td>
<td>Represents a legislated OBD-II Baud rate</td>
</tr>
<tr>
<td>TPOBDIAddress</td>
<td>Represents a legislated OBD-II address for ECUs</td>
</tr>
<tr>
<td>TPOBDIService</td>
<td>Represents an OBD-II Service</td>
</tr>
<tr>
<td>TPOBDIIParameter</td>
<td>Represents a PCAN-OBD-2 parameter to be read or set</td>
</tr>
<tr>
<td>TPOBDIIHwType</td>
<td>Represents the type of CAN hardware to be initialized</td>
</tr>
<tr>
<td>TPOBDIStatus</td>
<td>Represents an OBD-II status/error code</td>
</tr>
<tr>
<td>TPOBDIError</td>
<td>Represents an OBD-II Response codes</td>
</tr>
<tr>
<td>TPOBDIInfoDataType</td>
<td>Represents the type of returned data in TPOBDIInfoData (OBD service $09)</td>
</tr>
</tbody>
</table>

3.5.1 TPOBDIICANHandle

Represents a PCAN-OBD-2 channel handle.

**Syntax**

**C++**

```cpp
#define TPOBDIICANHandle WORD
```

**C++ / CLR**

```cpp
#define TPOBDIICANHandle System::Int16
```

**C#**

```csharp
using TPOBDIICANHandle = System.Int16;
```

**Visual Basic**

```vbnet
Imports TPOBDIICANHandle = System.Int16
```

**Remarks:** TPOBDIICANHandle is defined for the PCAN-OBD-2 API but it is identical to a TPOBDIICANHandle from PCAN-UDS API, a PCANTPCANHandle from PCAN-ISO-TP API or TPCANHandle from PCAN-Basic API.

**.NET Framework Programming Languages**

An alias is used to represent a Channel handle under Microsoft .NET in order to originate a homogeneity between all programming languages listed above.

Aliases are defined in the Peak.Can.Obdii namespace for C# and VB .NET. However, including a namespace does not include the defined aliases.

If it is wished to work with aliases, those must be copied to the working file, right after the inclusion of the Peak.Can.Obdii namespace. Otherwise, just use the native type, which in this case is a Byte.
C#

using System;
using Peak.Can.Obdii;
using TPOBDIIIPid = System.Int16; // Alias's declaration for System.Byte

Visual Basic

Imports System
Imports Peak.Can.Obdii
Imports TPOBDIIIPid = System.Int16 ' Alias' declaration for System.Byte

See also: PCAN-OBD-2 Handle Definitions on page 123

3.5.2 TPOBDIIIPid

Represents a Parameter Identification (PID), a parameter used by OBD-II service $01 and $02.

Syntax

C++
#define TPOBDIIIPid BYTE

C++ / CLR
#define TPOBDIIIPid System::Byte

C#
using TPOBDIIIPid = System.Byte;

Visual Basic
Imports TPOBDIIIPid = System.Byte

.NET Framework Programming Languages

An alias is used to represent a Parameter Identifier under Microsoft .NET in order to originate a homogeneity between all programming languages listed above.

Aliases are defined in the Peak.Can.Obdii namespace for C# and VB .NET. However, including a namespace does not include the defined aliases.

If it is wished to work with aliases, those must be copied to the working file, right after the inclusion of the Peak.Can.Obdii namespace. Otherwise, just use the native type, which in this case is a Byte.

C#
using System;
using Peak.Can.Obdii;
using TPOBDIIIPid = System.Byte; // Alias's declaration for System.Byte

Visual Basic
Imports System
Imports Peak.Can.Obdii
Imports TPOBDIIPid = System.Byte ' Alias' declaration for System.Byte

See also: OBDII_RequestCurrentData on page 110 (class-method: RequestCurrentData)
OBDII_RequestFreezeFrameData on page 111 (class-method: RequestFreezeFrameData)

3.5.3 TPOBDIIOBDMid
Represents an On-Board Diagnostic Monitor ID (OBDMID), a parameter used by OBD-II service $06.

Syntax
C++
#define TPOBDIIOBDMid BYTE

C++ / CLR
#define TPOBDIIOBDMid System::Byte

C# Syntax
using TPOBDIIOBDMid = System.Byte;

Visual Basic
Imports TPOBDIIOBDMid = System.Byte

.NET Framework Programming Languages

An alias is used to represent an OBDM Identifier under Microsoft .NET in order to originate a homogeneity between all programming languages listed above.

Aliases are defined in the Peak.Can.Obdii namespace for C# and VB .NET. However, including a namespace does not include the defined aliases.

If it is wished to work with aliases, those must be copied to the working file, right after the inclusion of the Peak.Can.Obdii namespace. Otherwise, just use the native type, which in this case is a Byte.

C#

using System;
using Peak.Can.Obdii;
using TPOBDIIOBDMid = System.Byte; // Alias's declaration for System.Byte

Visual Basic
Imports System
Imports Peak.Can.Obdii
Imports TPOBDIIOBDMid = System.Byte ' Alias' declaration for System.Byte

See also: OBDII_RequestTestResults on page 115 (class-method: RequestTestResults)

3.5.4 TPOBDIITid
Represents a Test ID (TID), a parameter used by OBD-II service $08.
**Syntax**

**C++**
```cpp
#define TPOBDIITid BYTE
```

**C++ / CLR**
```cpp
#define TPOBDIITid System::Byte
```

**C#**
```csharp
using TPOBDIITid = System.Byte;
```

**Visual Basic**
```vbnet
Imports TPOBDIITid = System.Byte
```

**.NET Framework Programming Languages**

An alias is used to represent a Test Identifier under Microsoft .NET in order to originate a homogeneity between all programming languages listed above.

Aliases are defined in the `Peak.Can.Obdii` namespace for C# and VB .NET. However, including a namespace does not include the defined aliases.

If it is wished to work with aliases, those must be copied to the working file, right after the inclusion of the `Peak.Can.Obdii` namespace. Otherwise, just use the native type, which in this case is a Byte.

**C#**
```csharp
using System;
using Peak.Can.Obdii;
using TPOBDIITid = System.Byte; // Alias's declaration for System.Byte
```

**Visual Basic**
```vbnet
Imports System
Imports Peak.Can.Obdii
Imports TPOBDIITid = System.Byte ' Alias' declaration for System.Byte
```

See also: [OBDII_RequestControlOperation](#) on page 118 (class-method: RequestControlOperation)

### 3.5.5 TPOBDIIInfoType

Represents an `InfoType`, a parameter used by OBD-II service $09.

**Syntax**

**C++**
```cpp
#define TPOBDIIInfoType BYTE
```

**C++ / CLR**
```cpp
#define TPOBDIIInfoType System::Byte
```
An alias is used to represent an InfoType identifier under Microsoft .NET in order to originate an homogeneity between all programming languages listed above.

Aliases are defined in the Peak.Can.Obdii namespace for C# and VB .NET. However, including a namespace does not include the defined aliases.

If it is wished to work with aliases, those must be copied to the working file, right after the inclusion of the Peak.Can.Obdii namespace. Otherwise, just use the native type, which in this case is a Byte.

### 3.5.6 TPOBDIIBaudrateInfo

Represents a legislated OBD-II Baud rate. According with the programming language, this type can be a group of defined values or an enumeration.
C#

```csharp
public enum TPOBDIIBaudrateInfo : byte
{
    POBDII_BAUDRATE_NON_LEGISLATED = 0x00,
    POBDII_BAUDRATE_250K = 0x01,
    POBDII_BAUDRATE_500K = 0x02,
    POBDII_BAUDRATE_AUTODETECT = 0xFF,
}
```

Pascal OO

```pascal
TPOBDIIBaudrateInfo = ( 
    POBDII_BAUDRATE_NON_LEGISLATED = $00,
    POBDII_BAUDRATE_250K = $01,
    POBDII_BAUDRATE_500K = $02,
    POBDII_BAUDRATE_AUTODETECT = $FF
);
```

Visual Basic

```vbnet
Public Enum TPOBDIIBaudrateInfo As Byte
    POBDII_BAUDRATE_NON_LEGISLATED = &H0
    POBDII_BAUDRATE_250K = &H1
    POBDII_BAUDRATE_500K = &H2
    POBDII_BAUDRATE_AUTODETECT = &HFF
End Enum
```

Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_BAUDRATE_NON_LEGISLATED</td>
<td>0</td>
<td>Non legislated-OBD baudrate (note: this is used only as a returned value of GetValue function with parameter POBDII_PARAM_BAUDRATE)</td>
</tr>
<tr>
<td>POBDII_BAUDRATE_250K</td>
<td>1</td>
<td>250 kBit/s</td>
</tr>
<tr>
<td>POBDII_BAUDRATE_500K</td>
<td>2</td>
<td>500 kBit/s</td>
</tr>
<tr>
<td>POBDII_BAUDRATE_AUTODETECT</td>
<td>0xFF (255)</td>
<td>Auto-detect OBD-II baudrate (note: used only with the Initialize function)</td>
</tr>
</tbody>
</table>

See also: [OBDII_Initialize](#) on page 103 (class-method: Initialize).

### 3.5.7 TPOBDIIAddress

Represents a legislated OBD-II address for ECUs. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

C++

```c
#define TPOBDIIAddress BYTE
#define POBDII_ECU_1 0x01
#define POBDII_ECU_2 0x02
#define POBDII_ECU_3 0x03
#define POBDII_ECU_4 0x04
#define POBDII_ECU_5 0x05
#define POBDII_ECU_6 0x06
#define POBDII_ECU_7 0x07
#define POBDII_ECU_8 0x08
#define POBDII_ECU_9 0x09
```
Pascal OO

```pascal
{$Z1}
TPOBDIIAddress = ( 
  POBDII_ECU_1 = $01,
  POBDII_ECU_2 = $02,
  POBDII_ECU_3 = $03,
  POBDII_ECU_4 = $04,
  POBDII_ECU_5 = $05,
  POBDII_ECU_6 = $06,
  POBDII_ECU_7 = $07,
  POBDII_ECU_8 = $08
);
```

C#

```csharp
public enum TPOBDIIAddress : byte
{
    POBDII_ECU_1 = 0x01,
    POBDII_ECU_2 = 0x02,
    POBDII_ECU_3 = 0x03,
    POBDII_ECU_4 = 0x04,
    POBDII_ECU_5 = 0x05,
    POBDII_ECU_6 = 0x06,
    POBDII_ECU_7 = 0x07,
    POBDII_ECU_8 = 0x08,
}
```

C++ / CLR

```cpp
public enum TPOBDIIAddress : Byte
{
    POBDII_ECU_1 = 0x01,
    POBDII_ECU_2 = 0x02,
    POBDII_ECU_3 = 0x03,
    POBDII_ECU_4 = 0x04,
    POBDII_ECU_5 = 0x05,
    POBDII_ECU_6 = 0x06,
    POBDII_ECU_7 = 0x07,
    POBDII_ECU_8 = 0x08,
};
```

Visual Basic

```vb
Public Enum TPOBDIIAddress As Byte
    POBDII_ECU_1 = &H1
    POBDII_ECU_2 = &H2
    POBDII_ECU_3 = &H3
    POBDII_ECU_4 = &H4
    POBDII_ECU_5 = &H5
    POBDII_ECU_6 = &H6
    POBDII_ECU_7 = &H7
    POBDII_ECU_8 = &H8
End Enum
```
Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_ECU_1</td>
<td>1</td>
<td>ECU #1</td>
</tr>
<tr>
<td>POBDII_ECU_2</td>
<td>2</td>
<td>ECU #2</td>
</tr>
<tr>
<td>POBDII_ECU_3</td>
<td>3</td>
<td>ECU #3</td>
</tr>
<tr>
<td>POBDII_ECU_4</td>
<td>4</td>
<td>ECU #4</td>
</tr>
<tr>
<td>POBDII_ECU_5</td>
<td>5</td>
<td>ECU #5</td>
</tr>
<tr>
<td>POBDII_ECU_6</td>
<td>6</td>
<td>ECU #6</td>
</tr>
<tr>
<td>POBDII_ECU_7</td>
<td>7</td>
<td>ECU #7</td>
</tr>
<tr>
<td>POBDII_ECU_8</td>
<td>8</td>
<td>ECU #8</td>
</tr>
<tr>
<td>POBDII_ECU_9</td>
<td>9</td>
<td>ECU #9</td>
</tr>
</tbody>
</table>

3.5.8 TPOBDIIService

Represents an OBD-II Service. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

C++

```cpp
#define TPOBDIIService BYTE
#define POBDII_SERVICE_01 0x01
#define POBDII_SERVICE_02 0x02
#define POBDII_SERVICE_03 0x03
#define POBDII_SERVICE_04 0x04
#define POBDII_SERVICE_06 0x06
#define POBDII_SERVICE_07 0x07
#define POBDII_SERVICE_08 0x08
#define POBDII_SERVICE_09 0x09
#define POBDII_SERVICE_0A 0x0A
```

Pascal OO

```pascal
{$Z1}
TPOBDIIService = ( 
  POBDII_SERVICE_01 = $01,
  POBDII_SERVICE_02 = $02,
  POBDII_SERVICE_03 = $03,
  POBDII_SERVICE_04 = $04,
  POBDII_SERVICE_06 = $06,
  POBDII_SERVICE_07 = $07,
  POBDII_SERVICE_08 = $08,
  POBDII_SERVICE_09 = $09,
  POBDII_SERVICE_0A = $0A
);
```

C#

```csharp
public enum TPOBDIIService : byte 
{
    POBDII_SERVICE_01 = 0x01,
    POBDII_SERVICE_02 = 0x02,
    POBDII_SERVICE_03 = 0x03,
    POBDII_SERVICE_04 = 0x04,
    POBDII_SERVICE_06 = 0x06,
    POBDII_SERVICE_07 = 0x07,
}
### Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_SERVICE_01</td>
<td>1</td>
<td>Service $01: RequestCurrentData</td>
</tr>
<tr>
<td>POBDII_SERVICE_02</td>
<td>2</td>
<td>Service $02: RequestFreezeFrameData</td>
</tr>
<tr>
<td>POBDII_SERVICE_03</td>
<td>3</td>
<td>Service $03: RequestStoredTroubleCodes</td>
</tr>
<tr>
<td>POBDII_SERVICE_04</td>
<td>4</td>
<td>Service $04: ClearTroubleCodes</td>
</tr>
<tr>
<td>POBDII_SERVICE_06</td>
<td>6</td>
<td>Service $06: RequestTestResults</td>
</tr>
<tr>
<td>POBDII_SERVICE_07</td>
<td>7</td>
<td>Service $07: RequestPendingTroubleCodes</td>
</tr>
<tr>
<td>POBDII_SERVICE_08</td>
<td>8</td>
<td>Service $08: RequestControlOperation</td>
</tr>
<tr>
<td>POBDII_SERVICE_09</td>
<td>9</td>
<td>Service $09: RequestVehicleInformation</td>
</tr>
<tr>
<td>POBDII_SERVICE_0A</td>
<td>0x0A</td>
<td>Service $0A: RequestPermanentTroubleCodes</td>
</tr>
</tbody>
</table>

**Remarks:** Service $05 is not supported for ISO 15765-4. The functionality of Service $05 is implemented in Service $06.

### 3.5.9 TPOBDIIParameter

Represents a PCAN-OBD-2 parameter to be read or set. According with the programming language, this type can be a group of defined values or an enumeration. With some exceptions, a channel must first be initialized before their parameters can be read or set.

### Syntax
### C++

```cpp
#define TPOBDIIParameter BYTE
#define POBDII_PARAM_LOGGING 0xB1
#define POBDII_PARAM_AVAILABLE_ECUS 0xB2
#define POBDII_PARAM_SUPPORTMASK_PIDS 0xB3
#define POBDII_PARAM_SUPPORTMASK_FFPIDS 0xB4
#define POBDII_PARAM_SUPPORTMASK_OBDMIDS 0xB5
#define POBDII_PARAM_SUPPORTMASK_TIDS 0xB6
#define POBDII_PARAM_SUPPORTMASK_INFOTYPES 0xB7
#define POBDII_PARAM_API_VERSION 0xB8
#define POBDII_PARAM_BAUDRATE 0xB9
#define POBDII_PARAM_CAN_ID 0xBA
#define POBDII_PARAM_DEBUG 0xE3
#define POBDII_PARAM_CHANNEL_CONDITION 0xE4
```

### Pascal OO

```pascal
{$Z1}
TPOBDIIParameter = (  
  POBDII_PARAM_LOGGING = $B1,  
  POBDII_PARAM_AVAILABLE_ECUS = $B2,  
  POBDII_PARAM_SUPPORTMASK_PIDS = $B3,  
  POBDII_PARAM_SUPPORTMASK_FFPIDS = $B4,  
  POBDII_PARAM_SUPPORTMASK_OBDMIDS = $B5,  
  POBDII_PARAM_SUPPORTMASK_TIDS = $B6,  
  POBDII_PARAM_SUPPORTMASK_INFOTYPES = $B7,  
  POBDII_PARAM_API_VERSION = $B8,  
  POBDII_PARAM_BAUDRATE = $B9,  
  POBDII_PARAM_CAN_ID = $BA,  
  POBDII_PARAM_DEBUG = $E3,  
  POBDII_PARAM_CHANNEL_CONDITION =$E4  
);```

### C#

```csharp
public enum TPOBDIIParameter : byte
{
    POBDII_PARAM_LOGGING = 0xB1,
    POBDII_PARAM_AVAILABLE_ECUS = 0xB2,
    POBDII_PARAM_SUPPORTMASK_PIDS = 0xB3,
    POBDII_PARAM_SUPPORTMASK_FFPIDS = 0xB4,
    POBDII_PARAM_SUPPORTMASK_OBDMIDS = 0xB5,
    POBDII_PARAM_SUPPORTMASK_TIDS = 0xB6,
    POBDII_PARAM_SUPPORTMASK_INFOTYPES = 0xB7,
    POBDII_PARAM_API_VERSION = 0xB8,
    POBDII_PARAM_BAUDRATE = 0xB9,
    POBDII_PARAM_CAN_ID = 0xBA,
    POBDII_PARAM_DEBUG = 0xE3,
    POBDII_PARAM_CHANNEL_CONDITION = 0xE4
}
```

### C++ / CLR

```csharp
public enum TPOBDIIParameter : Byte
{
    POBDII_PARAM_LOGGING = 0xB1,
    POBDII_PARAM_AVAILABLE_ECUS = 0xB2,
    POBDII_PARAM_SUPPORTMASK_PIDS = 0xB3,
    POBDII_PARAM_SUPPORTMASK_FFPIDS = 0xB4,
    POBDII_PARAM_SUPPORTMASK_OBDMIDS = 0xB5,
    POBDII_PARAM_SUPPORTMASK_TIDS = 0xB6,
    POBDII_PARAM_SUPPORTMASK_INFOTYPES = 0xB7,
    POBDII_PARAM_API_VERSION = 0xB8,
    POBDII_PARAM_BAUDRATE = 0xB9,
    POBDII_PARAM_CAN_ID = 0xBA,
    POBDII_PARAM_DEBUG = 0xE3,
    POBDII_PARAM_CHANNEL_CONDITION = 0xE4
}
```
PCAN-OBD-2 API – Documentation

POBDII_PARAM_SUPPORTMASK_FFPIDS = 0xB4,
POBDII_PARAM_SUPPORTMASK_OBDMIDS = 0xB5,
POBDII_PARAM_SUPPORTMASK_TIDS = 0xB6,
POBDII_PARAM_SUPPORTMASK_INFOTYPES = 0xB7,
POBDII_PARAM_API_VERSION = 0xB8,
POBDII_PARAM_BAUDRATE = 0xB9,
POBDII_PARAM_CAN_ID = 0xBA,
POBDII_PARAM_DEBUG = 0xE3,
POBDII_PARAM_CHANNEL_CONDITION = 0xE4

Visual Basic

Public Enum TPOBDIIParameter As Byte
    POBDII_PARAM_LOGGING = &HB1
    POBDII_PARAM_AVAILABLE_ECUS = &HB2
    POBDII_PARAM_SUPPORTMASK_PIDS = &HB3
    POBDII_PARAM_SUPPORTMASK_FFPIDS = &HB4
    POBDII_PARAM_SUPPORTMASK_OBDMIDS = &HB5
    POBDII_PARAM_SUPPORTMASK_TIDS = &HB6
    POBDII_PARAM_SUPPORTMASK_INFOTYPES = &HB7
    POBDII_PARAM_BAUDRATE = &HB9
    POBDII_PARAM_CAN_ID = &HBA
    POBDII_PARAM_API_VERSION = &HB8
    POBDII_PARAM_DEBUG = &HE3
    POBDII_PARAM_CHANNEL_CONDITION = &HE4
End Enum

Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_PARAM_LOGGING</td>
<td>177 (0xB1)</td>
<td>Byte</td>
<td>Logging mode</td>
</tr>
<tr>
<td>POBDII_PARAM_AVAILABLE_ECUS</td>
<td>178 (0xB2)</td>
<td>Byte</td>
<td>Number of available ECUs</td>
</tr>
<tr>
<td>POBDII_PARAM_SUPPORTMASK_PIDS</td>
<td>179 (0xB3)</td>
<td>Array of Byte (256)</td>
<td>Supported PIDs for Service 01: Current Data</td>
</tr>
<tr>
<td>POBDII_PARAM_SUPPORTMASK_FFPIDS</td>
<td>180 (0xB4)</td>
<td>Array of Byte (257)</td>
<td>Supported PIDs of Frame (identified by the first BYTE in the buffer) for Service 02: Freeze Frame Data</td>
</tr>
<tr>
<td>POBDII_PARAM_SUPPORTMASK_OBDMIDS</td>
<td>181 (0xB5)</td>
<td>Array of Byte (256)</td>
<td>Supported OBDMIDs for Service 06: Test Results</td>
</tr>
<tr>
<td>POBDII_PARAM_SUPPORTMASK_TIDS</td>
<td>182 (0xB6)</td>
<td>Array of Byte (256)</td>
<td>Supported TIDs for Service 08: Control Operation</td>
</tr>
<tr>
<td>POBDII_PARAM_SUPPORTMASK_INFOTYPES</td>
<td>183 (0xB7)</td>
<td>Array of Byte (256)</td>
<td>Supported InfoTypes for Service 09: Vehicle Information</td>
</tr>
<tr>
<td>POBDII_PARAM_BAUDRATE</td>
<td>185 (0xB9)</td>
<td>Byte</td>
<td>Baudrate of the channel (see TPOBDII_BaudrateInfo)</td>
</tr>
<tr>
<td>POBDII_PARAM_CAN_ID</td>
<td>186 (0xBA)</td>
<td>Byte</td>
<td>CAN identifier length (11 or 29 bits)</td>
</tr>
<tr>
<td>POBDII_PARAM_API_VERSION</td>
<td>184 (0xB8)</td>
<td>String</td>
<td>API version of the PCAN-OBD-2 API</td>
</tr>
<tr>
<td>POBDII_PARAM_DEBUG</td>
<td>227 (0xE3)</td>
<td>Byte</td>
<td>Debug mode</td>
</tr>
<tr>
<td>POBDII_PARAM_CHANNEL_CONDITION</td>
<td>228 (0xE4)</td>
<td>Byte</td>
<td>PCAN-OBD-2 channel condition</td>
</tr>
</tbody>
</table>
Characteristics

POBDII_PARAM_LOGGING

Access: \[ \mathbb{R} \cup \mathbb{W} \]

Description: This value is used to control logging mode.

Possible values: POBDII_LOGGING_NONE disables logging mode, POBDII_LOGGING_TO_FILE enables it and logs data to file, POBDII_LOGGING_TOSTDOUT enables it and logs data to the standard output.

Default value: POBDII_LOGGING_NONE.

PCAN-Device: All PCAN devices (excluding POBDII_NONEBUS channel).

POBDII_PARAM_AVAILABLE_ECUS

Access: \[ \mathbb{R} \]

Description: This value is used to get the number of known ECUs.

Possible values: A positive numeric value (0 to 8 for legislated-OBD-compliant vehicle).

Default value: NA.

PCAN-Device: All PCAN devices (excluding POBDII_NONEBUS channel).

POBDII_PARAM_SUPPORTMASK_PIDS

Access: \[ \mathbb{R} \]

Description: This value is used to retrieve the list of supported Parameter Identifiers (PIDs) by the connected ECUs.

Possible values: The list is represented by an array of 256 bytes where each row corresponds to the support of a specific PID (i.e. row 0 is PID #0, row 1 is PID 1, etc.). The values of the array are bit-encoded and each bit corresponds to an ECU where:

- bit #0 corresponds to ECU#1 and bit #7 to ECU#8,
- a bit set to 1 states that the PID is supported by the ECU

For instance, if the variable ‘array’ contains the list of supported identifiers and ‘array’[12] = 5, then PID#12 is supported by ECU #1 and #3.

Default value: NA.

PCAN-Device: All PCAN devices (excluding POBDII_NONEBUS channel).

POBDII_PARAM_SUPPORTMASK_FFPIDS

Access: \[ \mathbb{R} \]

Description: This value is used to retrieve the list of supported FreezeFrame Parameter Identifiers (FFPIDs) by the connected ECUs.

Possible values: The list is represented by an array of 257 bytes where the first byte is the Frame number indicator and each following row corresponds to the support of a specific FFPID (i.e. row 0 is the frame number, row 1 is FFPID #0, row 2 is FFPID #1, etc.). The values of the array are bit-encoded and each bit corresponds to an ECU where:
bit #0 corresponds to ECU #1 and bit #7 to ECU #8,
a bit set to 1 states that the FFPID is supported by the ECU

For instance, if the variable ‘array’ contains the list of supported identifiers and ‘array’[12] = 5, then FFPID#11 is supported by ECU #1 and #3.

Default value: NA.

PCAN-Device: All PCAN devices (excluding POBDII_NONEBUS channel).

POBDII_PARAM_SUPPORTMASK_OBDMIDS

Access: 

Description: This value is used to retrieve the list of On-Board Monitoring Identifiers (OBDMIDs) by the connected ECUs.

Possible values: The list is represented by an array of 256 bytes where each row corresponds to the support of a specific OBDMID (i.e. row 0 is OBDMID #0, row 1 is OBDMID #1, etc.). The values of the array are bit-encoded and each bit corresponds to an ECU where:

- bit #0 corresponds to ECU #1 and bit #7 to ECU #8,
- a bit set to 1 states that the OBDMID is supported by the ECU

For instance, if the variable ‘array’ contains the list of supported identifiers and ‘array’[12] = 5, then OBDMID#12 is supported by ECU #1 and #3.

Default value: NA.

PCAN-Device: All PCAN devices (excluding POBDII_NONEBUS channel).

POBDII_PARAM_SUPPORTMASK_TIDS

Access: 

Description: This value is used to retrieve the list of supported Test Identifiers (TIDs) by the connected ECUs.

Possible values: The list is represented by an array of 256 bytes where each row corresponds to the support of a specific TID (i.e. row 0 is TID #0, row 1 is TID #1, etc.). The values of the array are bit-encoded and each bit corresponds to an ECU where:

- bit #0 corresponds to ECU #1 and bit #7 to ECU #8,
- a bit set to 1 states that the TID is supported by the ECU

For instance, if the variable ‘array’ contains the list of supported identifiers and ‘array’[12] = 5, then TID#12 is supported by ECU #1 and #3.

Default value: NA.

PCAN-Device: All PCAN devices (excluding POBDII_NONEBUS channel).

POBDII_PARAM_SUPPORTMASK_INFOTYPES

Access: 

Description: This value is used to retrieve the list of supported InfoTypes by the connected ECUs.
Possible values: The list is represented by an array of 256 bytes where each row corresponds to the support of a specific INFO_TYPE (i.e. row 0 is INFO_TYPE #0, row 1 is INFO_TYPE #1, etc.). The values of the array are bit-encoded and each bit corresponds to an ECU where:

- bit #0 corresponds to ECU #1 and bit #7 to ECU #8,
- a bit set to 1 states that the INFO_TYPE is supported by the ECU

For instance, if the variable ‘array’ contains the list of supported identifiers and ‘array’[12] = 5, then INFO_TYPE #12 is supported by ECU #1 and #3.

Default value: NA.

PCAN-Device: All PCAN devices (excluding POBDII_NONEBUS channel).

POBDII_PARAM_BAUDRATE

Access: 

Description: This value is used to get the initialized or detected baudrate of the POBDII channel.

Possible values: POBDII_BAUDRATE_NON_LEGISLATED, POBDII_BAUDRATE_250K or POBDII_BAUDRATE_500K.

Default value: NA.

PCAN-Device: All PCAN devices (excluding POBDII_NONEBUS channel).

POBDII_PARAM_CAN_ID

Access: 

Description: This value is used to get the bit length of the CAN identifiers used during the communication with the ECUs.

Possible values: POBDII_CAN_ID_11BIT or POBDII_CAN_ID_29BIT.

Default value: NA.

PCAN-Device: All PCAN devices (excluding POBDII_NONEBUS channel).

POBDII_PARAM_API_VERSION

Access: 

Description: This parameter is used to get information about the PCAN-OBD-2 API implementation version.

Possible values: The value is a null-terminated string indicating the version number of the API implementation. The returned text has the following form: x,x,x,x for major, minor, release and build. It represents the binary version of the API, within two 32-bit integers, defined by four 16-bit integers. The length of this text value will have a maximum length of 24 bytes, 5 bytes for represent each 16-bit value, three separator characters ( , or . ) and the null-termination.

Default value: NA.

PCAN-Device: NA. Any PCAN device can be used, including the POBDII_NONEBUS channel.
POBDII_PARAM_DEBUG

Access: R W

Description: This parameter is used to control debug mode. If enabled, any received or transmitted CAN frames will be printed to the standard output.

Possible values: POBDII_DEBUG_NONE disables debug mode and POBDII_DEBUG_CAN enables it.

Default value: POBDII_DEBUG_NONE.

PCAN-Device: All PCAN devices (excluding POBDII_NONEBUS channel).

POBDII_PARAM_CHANNEL_CONDITION

Access: R

Description: This parameter is used to check and detect available PCAN hardware on a computer, even before trying to connect any of them. This is useful when an application wants the user to select which hardware should be using in a communication session.

Possible values: This parameter can have one of these values: POBDII_CHANNEL_UNAVAILABLE, POBDII_CHANNEL_AVAILABLE, and POBDII_CHANNEL_OCCUPIED.

Default value: NA.

PCAN-Device: All PCAN devices (excluding PCAN_NONEBUS channel).

Note: It is not needed to have a PCAN channel initialized before asking for its condition.

3.5.10 TPOBDIIHwType

Represents the type of CAN hardware to be initialized. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

C++

```cpp
#define TPOBDIIHWType BYTE
#define POBDII_HWTYPE_ISA 0x01
#define POBDII_HWTYPE_ISA_SJA 0x09
#define POBDII_HWTYPE_ISA_PHYTEC 0x04
#define POBDII_HWTYPE_DNG 0x02
#define POBDII_HWTYPE_DNG_EPP 0x03
#define POBDII_HWTYPE_DNG_SJA 0x05
#define POBDII_HWTYPE_DNG_SJA_EPP 0x06
```

Pascal OO

```pascal
{$Z1}
TPOBDIIHwType = (
  POBDII_HWTYPE_ISA = $01,
  POBDII_HWTYPE_ISA_SJA = $09,
  POBDII_HWTYPE_ISA_PHYTEC = $04,
  POBDII_HWTYPE_DNG = $02,
  POBDII_HWTYPE_DNG_EPP = $03,
  POBDII_HWTYPE_DNG_SJA = $05,
  POBDII_HWTYPE_DNG_SJA_EPP = $06
);```
C#

```csharp
public enum TPOBDIIHwType : byte
{
    POBDII_HWTYPE_ISA = 0x01,
    POBDII_HWTYPE_ISA_SJA = 0x09,
    POBDII_HWTYPE_ISA_PHYTEC = 0x04,
    POBDII_HWTYPE_DNG = 0x02,
    POBDII_HWTYPE_DNG_EPP = 0x03,
    POBDII_HWTYPE_DNG_SJA = 0x05,
    POBDII_HWTYPE_DNG_SJA_EPP = 0x06,
}
```

C++ / CLR

```c++
public enum TPOBDIIHwType : Byte
{
    POBDII_HWTYPE_ISA = 0x01,
    POBDII_HWTYPE_ISA_SJA = 0x09,
    POBDII_HWTYPE_ISA_PHYTEC = 0x04,
    POBDII_HWTYPE_DNG = 0x02,
    POBDII_HWTYPE_DNG_EPP = 0x03,
    POBDII_HWTYPE_DNG_SJA = 0x05,
    POBDII_HWTYPE_DNG_SJA_EPP = 0x06,
};
```

Visual Basic

```vbnet
Public Enum TPOBDIIHwType As Byte
    POBDII_HWTYPE_ISA = &H1
    POBDII_HWTYPE_ISA_SJA = &H9
    POBDII_HWTYPE_ISA_PHYTEC = &H4
    POBDII_HWTYPE_DNG = &H2
    POBDII_HWTYPE_DNG_EPP = &H3
    POBDII_HWTYPE_DNG_SJA = &H5
    POBDII_HWTYPE_DNG_SJA_EPP = &H6
End Enum
```

### Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_HWTYPE_ISA</td>
<td>1</td>
<td>PCAN-ISA 82C200</td>
</tr>
<tr>
<td>POBDII_HWTYPE_ISA_SJA</td>
<td>9</td>
<td>PCAN-ISA SJA1000</td>
</tr>
<tr>
<td>POBDII_HWTYPE_ISA_PHYTEC</td>
<td>4</td>
<td>PHYTEC ISA</td>
</tr>
<tr>
<td>POBDII_HWTYPE_DNG</td>
<td>2</td>
<td>PCAN-Dongle 82C200</td>
</tr>
<tr>
<td>POBDII_HWTYPE_DNG_EPP</td>
<td>3</td>
<td>PCAN-Dongle EPP 82C200</td>
</tr>
<tr>
<td>POBDII_HWTYPE_DNG_SJA</td>
<td>5</td>
<td>PCAN-Dongle SJA1000</td>
</tr>
<tr>
<td>POBDII_HWTYPE_DNG_SJA_EPP</td>
<td>6</td>
<td>PCAN-Dongle EPP SJA1000</td>
</tr>
</tbody>
</table>

**See also:** [POBDII_Initialize](#) (class-method: Initialize)

### 3.5.11 TPOBDIIStatus

Represents an OBD-II status/error code. According with the programming language, this type can be a group of defined values or an enumeration.

**Syntax**
C++

```cpp
#define TPOBDIIStatus DWORD
#define POBDII_ERROR_OK 0x00000
#define POBDII_ERROR_NOT_INITIALIZED 0x00001
#define POBDII_ERROR_ALREADY_INITIALIZED 0x00002
#define POBDII_ERROR_NO_MEMORY 0x00003
#define POBDII_ERROR_OVERFLOW 0x00004
#define POBDII_ERROR_TIMEOUT 0x00006
#define POBDII_ERROR_NO_MESSAGE 0x00007
#define POBDII_ERROR_WRONG_PARAM 0x00008
#define POBDII_ERROR_NOT_SUPPORTED 0x00009
#define POBDII_ERROR_PARSE_ERROR 0x0000A
#define POBDII_ERROR_BUSLIGHT 0x0000B
#define POBDII_ERROR_BUSHEAVY 0x0000C
#define POBDII_ERROR_BUSOFF 0x0000D
#define POBDII_ERROR_UNSUPPORTED_ECUS 0x0000E
#define POBDII_ERROR_CAN_ERROR 0x10000
```

Pascal OO

```pascal
{$Z4}
TPOBDIIStatus = (   
  POBDII_ERROR_OK = $00000,   
  POBDII_ERROR_NOT_INITIALIZED = $00001,   
  POBDII_ERROR_ALREADY_INITIALIZED = $00002,   
  POBDII_ERROR_NO_MEMORY = $00003,   
  POBDII_ERROR_OVERFLOW = $00004,   
  POBDII_ERROR_TIMEOUT = $00006,   
  POBDII_ERROR_NO_MESSAGE = $00007,   
  POBDII_ERROR_WRONG_PARAM = $00008,   
  POBDII_ERROR_NOT_SUPPORTED = $00009,   
  POBDII_ERROR_PARSE_ERROR = $0000A,   
  POBDII_ERROR_BUSLIGHT = $0000B,   
  POBDII_ERROR_BUSHEAVY = $0000C,   
  POBDII_ERROR_BUSOFF = $0000D,   
  POBDII_ERROR_UNSUPPORTED_ECUS = $0000E,   
  POBDII_ERROR_CAN_ERROR = $10000   
);
```

C#

```csharp
public enum TPOBDIIStatus : uint
{
    POBDII_ERROR_OK = 0x00000,
    POBDII_ERROR_NOT_INITIALIZED = 0x00001,
    POBDII_ERROR_ALREADY_INITIALIZED = 0x00002,
    POBDII_ERROR_NO_MEMORY = 0x00003,
    POBDII_ERROR_OVERFLOW = 0x00004,
    POBDII_ERROR_TIMEOUT = 0x00006,
    POBDII_ERROR_NO_MESSAGE = 0x00007,
    POBDII_ERROR_WRONG_PARAM = 0x00008,
    POBDII_ERROR_NOT_SUPPORTED = 0x00009,
    POBDII_ERROR_PARSE_ERROR = 0x0000A,
    POBDII_ERROR_BUSLIGHT = 0x0000B,
    POBDII_ERROR_BUSHEAVY = 0x0000C,
    POBDII_ERROR_BUSOFF = 0x0000D,
    POBDII_ERROR_UNSUPPORTED_ECUS = 0x0000E,
    POBDII_ERROR_CAN_ERROR = 0x10000,
}
```
### C++ / CLR

```csharp
public enum TPOBDIIStatus : UInt32
{
    POBDII_ERROR_OK                     = 0x00000,
    POBDII_ERROR_NOT_INITIALIZED        = 0x00001,
    POBDII_ERROR_ALREADY_INITIALIZED    = 0x00002,
    POBDII_ERROR_NO_MEMORY              = 0x00003,
    POBDII_ERROR_OVERFLOW               = 0x00004,
    POBDII_ERROR_TIMEOUT                = 0x00006,
    POBDII_ERROR_NO_MESSAGE             = 0x00007,
    POBDII_ERROR_WRONG_PARAM            = 0x00008,
    POBDII_ERROR_NOT_SUPPORTED          = 0x00009,
    POBDII_ERROR_PARSE_ERROR  = 0x0000A,
    POBDII_ERROR_BUSLIGHT   = 0x0000B,
    POBDII_ERROR_BUSHEAVY   = 0x0000C,
    POBDII_ERROR_BUSOFF    = 0x0000D,
    POBDII_ERROR_UNSUPPORTED_ECUS      = 0x0000E,
    POBDII_ERROR_CAN_ERROR   = 0x10000,
};
```

### Visual Basic

```vba
Public Enum TPOBDIIStatus : UInt32
    POBDII_ERROR_OK = &H0
    POBDII_ERROR_NOT_INITIALIZED = &H1
    POBDII_ERROR_ALREADY_INITIALIZED = &H2
    POBDII_ERROR_NO_MEMORY = &H3
    POBDII_ERROR_OVERFLOW = &H4
    POBDII_ERROR_TIMEOUT = &H6
    POBDII_ERROR_NO_MESSAGE = &H7
    POBDII_ERROR_WRONG_PARAM = &H8
    POBDII_ERROR_NOT_SUPPORTED = &H9
    POBDII_ERROR_PARSE_ERROR = &HA
    POBDII_ERROR_BUSLIGHT = &HB
    POBDII_ERROR_BUSHEAVY = &HC
    POBDII_ERROR_BUSOFF = &HD
    POBDII_ERROR_UNSUPPORTED_ECUS = &HE
    POBDII_ERROR_CAN_ERROR = &H10000
End Enum
```

**Remarks:** The `POBDII_ERROR_CAN_ERROR` status is a generic error code that is used to identify PCAN-Basic errors (as PCAN-Basic API is used internally by the PCAN-OBD-2 API). When a PCAN-Basic error occurs, the API performs a bitwise combination of the `POBDII_ERROR_CAN_ERROR` and the PCAN-Basic (`TPCANStatus`) error.

### Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_ERROR_OK</td>
<td>0x00000</td>
<td>No error. Success</td>
</tr>
<tr>
<td>POBDII_ERROR_NOT_INITIALIZED</td>
<td>0x00001</td>
<td>Not initialized</td>
</tr>
<tr>
<td>POBDII_ERROR_ALREADY_INITIALIZED</td>
<td>0x00002</td>
<td>Already initialized</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MEMORY</td>
<td>0x00003</td>
<td>Failed to allocate memory</td>
</tr>
<tr>
<td>POBDII_ERROR_OVERFLOW</td>
<td>0x00004</td>
<td>Buffer overflow occurred (too many channels initialized or too many messages in queue)</td>
</tr>
<tr>
<td>POBDII_ERROR_TIMEOUT</td>
<td>0x00006</td>
<td>Timeout while trying to access the PCAN-OBD-2 API</td>
</tr>
<tr>
<td>Name</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MESSAGE</td>
<td>0x000007 (000007)</td>
<td>No message available</td>
</tr>
<tr>
<td>POBDII_ERROR_WRONG_PARAM</td>
<td>0x000008 (000008)</td>
<td>Invalid parameter</td>
</tr>
<tr>
<td>POBDII_ERROR_NOT_SUPPORTED</td>
<td>0x000009 (000009)</td>
<td>The OBDII response contains vehicle specific parameters that are not supported</td>
</tr>
<tr>
<td>POBDII_ERROR_PARSE_ERROR</td>
<td>0x0000A0 (00010)</td>
<td>Failed to parse OBDII response</td>
</tr>
<tr>
<td>POBDII_ERROR_BUSLIGHT</td>
<td>0x0000B0 (00011)</td>
<td>Bus error: an error counter reached the 'light' limit</td>
</tr>
<tr>
<td>POBDII_ERROR_BUSHEAVY</td>
<td>0x0000C0 (00012)</td>
<td>Bus error: an error counter reached the 'heavy' limit</td>
</tr>
<tr>
<td>POBDII_ERROR_BUSOFF</td>
<td>0x0000D0 (00013)</td>
<td>Bus error: the CAN controller is in bus-off state</td>
</tr>
<tr>
<td>POBDII_ERROR_UNSUPPORTED_ECUS</td>
<td>0x0000E0 (00014)</td>
<td>No connected ECUs, or ECU found is not supported</td>
</tr>
<tr>
<td>POBDII_ERROR_CAN_ERROR</td>
<td>0x80000000 (2147483648)</td>
<td>PCAN-Basic error flag (remove the flag to get a TPCANStatus error code)</td>
</tr>
</tbody>
</table>

3.5.12 **TPOBDIIError**

Represents an OBD-II Response codes. According with the programming language, this type can be a group of defined values or an enumeration.

**Syntax**

**C++**

```cpp
#define TPOBDIIError BYTE
#define POBDII_R_NO_ERROR                 0x00
#define POBDII_R_BUSY_REPEAT_REQUEST      0x21
#define POBDII_R_CONDITIONS_NOT_CORRECT   0x22
#define POBDII_R_RESPONSE_PENDING         0x78
#define POBDII_R_NOT_USED                 0xFF
```

**Pascal OO**

```pascal
{$Z1}
TPOBDIIError = (  
    POBDII_R_NO_ERROR = 0,
    POBDII_R_BUSY_REPEAT_REQUEST = $21,
    POBDII_R_CONDITIONS_NOT_CORRECT = $22,
    POBDII_R_RESPONSE_PENDING = $78,
    POBDII_R_NOT_USED = $FF
);  
```

**C#**

```csharp
public enum TPOBDIIError : byte
{
    POBDII_R_NO_ERROR = 0,
    POBDII_R_BUSY_REPEAT_REQUEST = 0x21,
    POBDII_R_CONDITIONS_NOT_CORRECT = 0x22,
    POBDII_R_RESPONSE_PENDING = 0x78,
    POBDII_R_NOT_USED = 0xFF,
}
```
public enum TPOBDIIError : Byte
{
    POBDII_R_NO_ERROR = 0,
    POBDII_R_BUSY_REPEAT_REQUEST = 0x21,
    POBDII_R_CONDITIONS_NOT_CORRECT = 0x22,
    POBDII_R_RESPONSE_PENDING = 0x78,
    POBDII_R_NOT_USED = 0xFF,
};

Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_R_NO_ERROR</td>
<td>0</td>
<td>No error, Positive response</td>
</tr>
<tr>
<td>POBDII_R_BUSY_REPEAT_REQUEST</td>
<td>0x21</td>
<td>Server is busy</td>
</tr>
<tr>
<td>POBDII_R_CONDITIONS_NOT_CORRECT</td>
<td>0x22</td>
<td>Conditions not correct</td>
</tr>
<tr>
<td>POBDII_R_RESPONSE_PENDING</td>
<td>0x78</td>
<td>Server needs more time</td>
</tr>
<tr>
<td>POBDII_R_NOT_USED</td>
<td>0xFF</td>
<td>Not a response, invalid value</td>
</tr>
</tbody>
</table>

3.5.13 TPOBDIIInfoDataType

Represents the type of returned data in TPOBDIIInfoData (OBD service $09). According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

C++

```c++
#define TPOBDIIInfoDataType BYTE
#define POBDII_INFOTYPE_COUNTER 0x00
#define POBDII_INFOTYPE_CALDATA 0x01
#define POBDII_INFOTYPE_STRING 0x02
#define POBDII_INFOTYPE_NONE 0x03
```

Pascal OO

```pascal
{$Z1}
TPOBDIIInfoDataType = (
    POBDII_INFOTYPE_COUNTER = $00,
    POBDII_INFOTYPE_CALDATA = $01,
    POBDII_INFOTYPE_STRING = $02,
    POBDII_INFOTYPE_NONE = $03
);
```
C#

```csharp
public enum TPOBDIIInfoDataType : byte
{
    POBDII_INFOTYPE_COUNTER = 0x00,
    POBDII_INFOTYPE_CALDATA = 0x01,
    POBDII_INFOTYPE_STRING = 0x02,
    POBDII_INFOTYPE_NONE = 0x03
}
```

C++ / CLR

```cpp
public enum TPOBDIIInfoDataType : Byte
{
    POBDII_INFOTYPE_COUNTER = 0x00,
    POBDII_INFOTYPE_CALDATA = 0x01,
    POBDII_INFOTYPE_STRING = 0x02,
    POBDII_INFOTYPE_NONE = 0x03
};
```

Visual Basic

```vbnet
Public Enum TPOBDIIInfoDataType As Byte
    POBDII_INFOTYPE_COUNTER = &H0
    POBDII_INFOTYPE_CALDATA = &H1
    POBDII_INFOTYPE_STRING = &H2
    POBDII_INFOTYPE_NONE = &H3
End Enum
```

Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_INFOTYPE_COUNTER</td>
<td>0</td>
<td>Data is a numeric value (2 bytes)</td>
</tr>
<tr>
<td>POBDII_INFOTYPE_CALDATA</td>
<td>1</td>
<td>Data corresponds to Calibration data (4 bytes)</td>
</tr>
<tr>
<td>POBDII_INFOTYPE_STRING</td>
<td>2</td>
<td>Data is a string (buffer containing up to 20 characters)</td>
</tr>
<tr>
<td>POBDII_INFOTYPE_NONE</td>
<td>3</td>
<td>Data is not an InfoType, this type is used when response contains bit encoded data describing InfoType Support</td>
</tr>
</tbody>
</table>
3.6 Methods

The methods defined for the classes OBDIIApi and TobdiiApi are divided in 4 groups of functionality. Note that these methods are static and can be called in the name of the class, without instantiation.

### Connection

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initializes a POBDII Channel</td>
</tr>
<tr>
<td>Uninitialize</td>
<td>Uninitializes a POBDII Channel</td>
</tr>
</tbody>
</table>

### Configuration

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetValue</td>
<td>Sets a configuration or information value within a POBDII Channel</td>
</tr>
</tbody>
</table>

### Information

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetValue</td>
<td>Retrieves information from a POBDII Channel</td>
</tr>
<tr>
<td>GetStatus</td>
<td>Retrieves the current BUS status of a POBDII Channel</td>
</tr>
<tr>
<td>GetUnitAndScaling</td>
<td>Retrieves the unit and scaling information for a specified Unit and Scaling ID</td>
</tr>
</tbody>
</table>

### Communication

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Resets the receive and transmit queues of a POBDII Channel</td>
</tr>
<tr>
<td>RequestCurrentData</td>
<td>Sends an OBDII Service $01 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>RequestFreezeFrameData</td>
<td>Sends an OBDII Service $02 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>RequestStoredTroubleCodes</td>
<td>Sends an OBDII Service $03 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>ClearTroubleCodes</td>
<td>Sends an OBDII Service $04 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>RequestTestResults</td>
<td>Sends an OBDII Service $06 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>RequestPendingTroubleCodes</td>
<td>Sends an OBDII Service $07 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>RequestControlOperation</td>
<td>Sends an OBDII Service $08 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>RequestVehicleInformation</td>
<td>Sends an OBDII Service $09 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>RequestPermanentTroubleCodes</td>
<td>Sends an OBDII Service $0A request into queue and waits to receive the responses</td>
</tr>
</tbody>
</table>

#### 3.6.1 Initialize

Initializes a POBDII Channel.

### Overloads

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize (TPOBDIICANHandle)</td>
<td>Initializes a POBDII Channel using baudrate autodetection</td>
</tr>
<tr>
<td>Initialize (TPOBDIICANHandle, TPOBDIIHwType, UlInt32, UlInt16)</td>
<td>Initializes a POBDII Channel</td>
</tr>
</tbody>
</table>
3.6.2 Initialize (TPOBDIICANHandle)

Initializes a POBDII Channel which represents a Plug & Play PCAN-Device using a baudrate autodetection mechanism.

Syntax

**Pascal OO**

```pascal
class function Initialize(
    CanChannel: TPOBDIICANHandle
): TPOBDIStatus; overload;
```

**C#**

```csharp
public static TPOBDIStatus Initialize(
    TPOBDIICANHandle CanChannel);
```

**C++ / CLR**

```cpp
static TPOBDIStatus Initialize(
    TPOBDIICANHandle CanChannel);
```

**Visual Basic**

```vb
Public Shared Function Initialize(
    ByVal CanChannel As TPOBDIICANHandle) As TPOBDIStatus
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
</tbody>
</table>

Returns

The return value is a [TPOBDIStatus] code. [POBDII_ERROR_OK] is returned on success. The typical errors in case of failure are:

- [POBDII_ERROR_ALREADY_INITIALIZED] Indicates that the desired POBDII Channel is already in use
- [POBDII_ERROR_UNSUPPORTED_ECUS] Indicates that no ECUs were found during the autodetection mechanism
- [POBDII_ERROR_OVERFLOW] Maximum number of initialized channels reached
- [POBDII_ERROR_CAN_ERROR] This error flag states that the error is composed of a more precise PCAN-Basic error

Remarks: As indicated by its name, the Initialize method initiates a POBDII Channel, preparing it for communication within the CAN bus connected to it. Calls to the other methods will fail if they are used with a Channel handle, different than [POBDII_NONEBUS], that has not been initialized yet. Each initialized channel should be released when it is not needed anymore.

Initializing a POBDII Channel means:

- to reserve the Channel for the calling application/process
- to detect the baudrate of the CAN bus
- to allocate channel resources, like receive and transmit queues
- to forward initialization to PCAN-UDS, PCAN-ISO-TP API, and PCAN-Basic API, hence registering/connecting the Hardware denoted by the channel handle

The Initialization process will fail if an application tries to initialize a POBDII-Channel that has already been initialized within the same process.
Take into consideration that initializing a channel causes a reset of the CAN hardware. In this way errors like BUSOFF, BUSHEAVY, and BUSLIGHT, are removed.

**Example**
The following example shows the initialize and uninitialize processes for a Plug-And-Play channel (channel 2 of a PCAN-PCI hardware).

**C#**

```csharp
TPOBDIIStatus result;

// The Plug & Play Channel (PCAN-PCI) is initialized
result = OBDIIApi.Initialize(OBDIIApi.POBDII_PCIBUS2);
if (result != TPOBDIIStatus.POBDII_ERROR_OK)
    MessageBox.Show("Initialization failed");
else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized");

// All initialized channels are released
OBDIIApi.Uninitialize(OBDIIApi.POBDII_NONEBUS);
```

**C++ / CLR**

```c++
TPOBDIIStatus result;

// The Plug & Play Channel (PCAN-PCI) is initialized
result = OBDIIApi::Initialize(OBDIIApi::POBDII_PCIBUS2);
if (result != POBDII_ERROR_OK)
    MessageBox::Show("Initialization failed");
else
    MessageBox::Show("PCAN-PCI (Ch-2) was initialized");

// All initialized channels are released
OBDIIApi::Uninitialize(OBDIIApi::POBDII_NONEBUS);
```

**Visual Basic**

```vbnet
Dim result As TPOBDIIStatus

' The Plug & Play Channel (PCAN-PCI) is initialized
result = OBDIIApi.Initialize(OBDIIApi.POBDII_PCIBUS2)
If result <> TPOBDIIStatus.POBDII_ERROR_OK Then
    MessageBox.Show("Initialization failed")
Else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized")
End If

' All initialized channels are released
OBDIIApi.Uninitialize(OBDIIApi.POBDII_NONEBUS);
```

**Pascal OO**

```pascal
var
    result: TPOBDIIStatus;

begin
    // The Plug & Play Channel (PCAN-PCI) is initialized
    result := TObdiiApi.Initialize(TObdiiApi.POBDII_PCIBUS2);
```
if (result <> POBDII_ERROR_OK) then
    MessageBox(0, 'Initialization failed', 'Error', MB_OK)
else
    MessageBox(0, 'PCAN-PCI (Ch-2) was initialized', 'Success', MB_OK);

// All initialized channels are released
TObdiiApi.Uninitialize(TObdiiApi.POBDII_NONEBUS);
End;

See also: [Uninitialize] on page 50, [GetValue] on page 55, Understanding PCAN-OBD-2 on page 6

Plain function version: [OBDII_Initialize]

3.6.3 Initialize (TPBODIICANHandle, TPBODIIBaudrateInfo, TPBODIIHwType, UInt32, UInt16)

Initializes a POBDII Channel.

Syntax

Pascal OO

class function Initialize(
    CanChannel: TPOBDIICANHandle;
    Baudrate: TPBODIIBaudrateInfo;
    HwType: TPBODIIHwType;
    IOPort: LongWord;
    Interrupt: Word): TPOBDIStatus; overload;

C#

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_Initialize")]
public static extern TPOBDIIStatus Initialize(
    [MarshalAs(UnmanagedType.U2)]
    TPOBDIICANHandle CanChannel,
    [MarshalAs(UnmanagedType.U1)]
    TPBODIIBaudrateInfo Baudrate,
    [MarshalAs(UnmanagedType.U1)]
    TPBODIIHwType HwType,
    UInt32 IOPort,
    UInt16 Interrupt);

C++ / CLR

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_Initialize")]
static TPOBDIIStatus Initialize(
    [MarshalAs(UnmanagedType::U2)]
    TPOBDIICANHandle CanChannel,
    [MarshalAs(UnmanagedType::U1)]
    TPBODIIBaudrateInfo Baudrate,
    [MarshalAs(UnmanagedType::U1)]
    TPBODIIHwType HwType,
    UInt32 IOPort,
    UInt16 Interrupt);

Visual Basic

<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_Initialize")> _
Public Shared Function Initialize(  
    ByVal CanChannel As TPOBDIICANHandle,  
    ByVal Baudrate As TPOBDIIBaudrateInfo,  
    ByVal HwType As TPOBDIIHwType,  
    ByVal IOPort As UInt32,  
    ByVal Interrupt As UInt16) As TPOBDIIStatus
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Baudrate</td>
<td>The speed for the communication (see TPOBDIIBaudrateInfo on page 27). The speed is limited to legislated-OBD baudrate or the autodetection value</td>
</tr>
<tr>
<td>HwType</td>
<td>The type of hardware (see TPOBDIIHwType on page 37)</td>
</tr>
<tr>
<td>IOPort</td>
<td>The I/O address for the parallel port</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Interrupt number of the parallel port</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

- POBDII_ERROR_ALREADY_INITIALIZED Indicates that the desired POBDII Channel is already in use
- POBDII_ERROR_UNSUPPORTED_ECUS Indicates that no ECUs were found during the autodetection mechanism
- POBDII_ERROR_OVERFLOW Maximum number of initialized channels reached
- POBDII_ERROR_CAN_ERROR This error flag states that the error is composed of a more precise PCAN-Basic error

Remarks: As indicated by its name, the Initialize method initiates a POBDII Channel, preparing it for communication within the CAN bus connected to it. Calls to the other methods will fail if they are used with a Channel handle, different than POBDII_NONEBUS, that has not been initialized yet. Each initialized channel should be released when it is not needed anymore.

Initializing a POBDII Channel means:

- to reserve the Channel for the calling application/process
- to detect the baudrate of the CAN bus (if requested with TPOBDIIBaudrateInfo, POBDII_BAUDRATE_AUTODETECT value)
- to allocate channel resources, like receive and transmit queues
- to forward initialization to PCAN-UDS, PCAN-ISO-TP API, and PCAN-Basic API, hence registering/connecting the Hardware denoted by the channel handle

The Initialization process will fail if an application tries to initialize a POBDII-Channel that has already been initialized within the same process.

Take in consideration that initializing a channel causes a reset of the CAN hardware. In this way errors like BUSOFF, BUSHEAVY, and BUSLIGHT, are removed.

Example

The following example shows the initialize and uninitialize processes for a Non-Plug-And-Play channel (channel 1 of the PCAN-DNG).

**Note:** The initialization specifies a baudrate thus skipping the autodetection mechanism.
C#

```csharp
TPOBDIIStatus result;

// The Non-Plug & Play Channel (PCAN-DNG) is initialized
result = OBDIIApi.Initialize(OBDIIApi.POBDII_DNGBUS1,
                            TPOBDIIInfo.INTERFACE_DNGBUS1,
                            POBDII_BAUDRATE_500K,
                            POBDII_HWTYPE_DNGBUS1,
                            0x378, 7);
if (result != TPOBDIIStatus.POBDII_ERROR_OK)
    MessageBox.Show("Initialization failed");
else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized");

// All initialized channels are released
OBDIIApi.Uninitialize(OBDIIApi.POBDII_NONEBUS);
```

C++ / CLR

```cpp
TPOBDIIStatus result;

// The Non-Plug & Play Channel (PCAN-DNG) is initialized
result = OBDIIApi::Initialize(OBDIIApi::POBDII_DNGBUS1, POBDII_BAUDRATE_500K,
                              POBDII_HWTYPE_DNGBUS1, 0x378, 7);
if (result != POBDII_ERROR_OK)
    MessageBox::Show("Initialization failed");
else
    MessageBox::Show("PCAN-PCI (Ch-2) was initialized");

// All initialized channels are released
OBDIIApi::Uninitialize(OBDIIApi::POBDII_NONEBUS);
```

Visual Basic

```vbs
Dim result As TPOBDIIStatus

' The Non-Plug & Play Channel (PCAN-DNG) is initialized
result = OBDIIApi.Initialize(OBDIIApi.POBDII_DNGBUS1,
                            TPOBDII_BAUDRATE_500K,
                            POBDII_HWTYPE_DNGBUS1,
                            &H378, 7)
If result <> TPOBDIIStatus.POBDII_ERROR_OK Then
    MessageBox.Show("Initialization failed")
Else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized")
End If

' All initialized channels are released
OBDIIApi.Uninitialize(OBDIIApi.POBDII_NONEBUS)
```

Pascal OO

```pascalk
var
    result: TPOBDIIStatus;

begin
    // The Non-Plug & Play Channel (PCAN-PIC) is initialized
    result := TOBDIIApi.Initialize(TOBDIIApi.POBDII_DNGBUS1,
                                    POBDII_BAUDRATE_500K,
                                    POBDII_HWTYPE_DNGBUS1, $378, 7);
    if (result <> POBDII_ERROR_OK) then
        MessageBox(0, 'Initialization failed', 'Error', MB_OK)
end
```

---

49
else
    MessageBox(0, 'PCAN-PCI (Ch-2) was initialized', 'Error', MB_OK);

// All initialized channels are released
TobdiiApi.Uninitialize(TObdiiApi.POBDII_NONEBUS);
end;

See also: Uninitialize below, GetValue on page 55, Understanding PCAN-OBD-2 on page 6

Plain function version: OBDII_Initialize

3.6.4 Uninitialize
Uninitializes a POBDII Channel.

Syntax
Pascal OO

```pascal
class function Uninitialize(
    CanChannel: TPOBDIICANHandle
): TPOBDIIStatus;
```

C#

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_Uninitialize")]
public static extern TPOBDIIStatus Uninitialize(
    [MarshalAs(UnmanagedType.U2)]
    TPOBDIICANHandle CanChannel);
```

C++ / CLR

```cpp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_Uninitialize")]
static TPOBDIIStatus Uninitialize(
    [MarshalAs(UnmanagedType::U2)]
    TPOBDIICANHandle CanChannel);
```

Visual Basic

```vbnet
<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_Uninitialize")> _
Public Shared Function Uninitialize( _
    ByVal CanChannel As TPOBDIICANHandle) As TPOBDIIStatus
End Function
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

| POBDII_ERROR_NOT_INITIALIZED | Indicates that the given POBDII channel cannot be uninitialized because it was not found in the list of reserved channels of the calling application |

Remarks: A POBDII Channel can be released using one of this possibilities:
**Single-Release:** Given a handle of a POBDII channel initialized before with the method initialize. If the given channel can not be found then an error is returned

**Multiple-Release:** Giving the handle value `POBDII_NONEBUS` which instructs the API to search for all channels initialized by the calling application and release them all. This option cause no errors if no hardware were uninitialized

**Example**

The following example shows the initialize and uninitialize processes for a Plug-And-Play channel (channel 2 of a PCAN-PCI hardware).

**C#**

```csharp
tpobdiiestatus result;

// The Plug & Play Channel (PCAN-PCI) is initialized
result = OBDIIApi.Initialize(OBDIIApi.POBDII_PCIBUS2, TPOBDII_BaudrateInfo.POBDII_BAUDRATE_500K, (TPOBDIIHwType)0, 0, 0);
if (result != TPOBDIIStatus.POBDII_ERROR_OK)
    MessageBox.Show("Initialization failed");
else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized");

// Release channel
OBDIIApi.Uninitialize(OBDIIApi.POBDII_PCIBUS2);
```

**C++ / CLR**

```c++
tpobdiiestatus result;

// The Plug & Play Channel (PCAN-PCI) is initialized
result = OBDIIApi::Initialize(OBDIIApi::POBDII_PCIBUS2, POBDII_BAUDRATE_500K,
(TPOBDIIHwType) 0, 0, 0);
if (result != POBDII_ERROR_OK)
    MessageBox::Show("Initialization failed");
else
    MessageBox::Show("PCAN-PCI (Ch-2) was initialized");

// Release channel
OBDIIApi::Uninitialize(OBDIIApi::POBDII_PCIBUS2);
```

**Visual Basic**

```vb
Dim result As TPOBDIIStatus

' The Plug & Play Channel (PCAN-PCI) is initialized
result = OBDIIApi.Initialize(OBDIIApi.POBDII_PCIBUS2, TPOBDII_BaudrateInfo.POBDII_BAUDRATE_500K, 0, 0, 0)
If result <> TPOBDIIStatus.POBDII_ERROR_OK Then
    MessageBox.Show("Initialization failed")
Else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized")
End If

' Release channel
OBDIIApi.Uninitialize(OBDIIApi.POBDII_PCIBUS2)
```
Pascal OO

```pascal
var
    result: TPOBDIIStatus;

begin
    // The Plug & Play Channel (PCAN-PCI) is initialized
    result := TObdiiApi.Initialize(TObdiiApi.POBDII_PCIBUS2, POBDII_BAUDRATE_500K, TPOBDIIHwType(0), $378, 7);
    if (result <> POBDII_ERROR_OK) then
        MessageBox(0, 'Initialization failed', 'Error', MB_OK)
    else
        MessageBox(0, 'PCAN-PCI (Ch-2) was initialized', 'Error', MB_OK);

    // Release channel
    TObdiiApi.Uninitialize(TObdiiApi.POBDII_PCIBUS2);
end;
```

See also: [Initialize on page 44](#)

Plain function version: [OBDII_Uninitialize](#)

### 3.6.5 SetValue

Set a configuration or information numeric value within a POBDII Channel.

**Syntax**

**Pascal OO**

```pascal
class function SetValue(
    CanChannel: TPOBDIICANHandle;
    Parameter: TPOBDIIParameter;
    NumericBuffer: PLongWord;
    BufferLength: LongWord
) : TPOBDIIStatus; overload;
```

**C#**

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_SetValue")]
public static extern TPOBDIIStatus SetValue(
    [MarshalAs(UnmanagedType.U1)] TPOBDIICANHandle CanChannel,
    [MarshalAs(UnmanagedType.U1)] TPOBDIIParameter Parameter,
    ref UInt32 NumericBuffer,
    UInt32 BufferLength);
```
### C++ / CLR

```c++
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_SetValue")]
static TPOBDIIStatus SetValue(
    [MarshalAs(UnmanagedType::U1)]
    TPOBDIICANHandle CanChannel,
    [MarshalAs(UnmanagedType::U1)]
    TPOBDIIParameter Parameter,
    UInt32% NumericBuffer,
    UInt32 BufferLength);
```

### Visual Basic

```vbnet
<DllImport("PCAN-OBDII.dll",EntryPoint:"OBDII_SetValue")>  _
Public Shared Function SetValue(  _
    ByVal CanChannel As TPOBDIICANHandle,  _
    ByVal Parameter As TPOBDIIParameter,  _
    ByRef NumericBuffer As UInt32,  _
    ByVal BufferLength As UInt32) As TPOBDIIStatus
End Function
```

---

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the value to be set (see TPOBDIIParameter on page 31)</td>
</tr>
<tr>
<td>NumericBuffer</td>
<td>The buffer containing the numeric value to be set</td>
</tr>
<tr>
<td>BufferLength</td>
<td>The length in bytes of the given buffer</td>
</tr>
</tbody>
</table>

### Returns

The return value is a `TPOBDIIStatus` code. `POBDII_ERROR_OK` is returned on success. The typical errors in case of failure are:

- **POBDII_ERROR_NOT_INITIALIZED**
  Indicates that the given POBDII channel cannot be uninitialzied because it was not found in the list of reserved channels of the calling application

- **POBDII_ERROR_WRONG_PARAM**
  Indicates that the parameters passed to the method are invalid. Check the value of 'Parameter' and assert it is compatible with an integer buffer

### Remarks

Use the method `SetValue` to set configuration information or environment values of a POBDII Channel. Note that any calls with non OBDII parameters (i.e. `TPOBDIIParameter`) will be forwarded to PCAN-UDS, PCAN-ISO-TP API, and PCAN-Basic API.

More information about the parameters and values that can be set can be found in Parameter Value Definitions.

### Example

The following example shows the use of the method `SetValue` on the channel `POBDII_PCIBUS2` to enable debug mode.

**Note:** It is assumed that the channel was already initialized.

#### C#

```csharp
TPOBDIIStatus result;
UInt32 iBuffer = 0;
```
// Enable CAN DEBUG mode
iBuffer = OBDDIIapi.POBDII_DEBUG_CAN;
result = OBDDIIapi.SetValue(OBDDIIapi.POBDII_PCIBUS2, 
    TOBDDIIParameter.POBDII_PARAM_DEBUG, ref iBuffer, sizeof(UInt32));
if (result != TPOBDIIStatus.POBDII_ERROR_OK)
    MessageBox.Show("Failed to set value");
else
    MessageBox.Show("Value changed successfully ");

C++/CLR

TPOBDIIStatus result;
UInt32 iBuffer = 0;
// Enable CAN DEBUG mode
iBuffer = OBDIIApi::POBDII_DEBUG_CAN;
result = OBDIIApi::SetValue(OBDIIApi::POBDII_PCIBUS2, POBDII_PARAM_DEBUG, iBuffer, 
    sizeof(UInt32));
if (result != POBDII_ERROR_OK)
    MessageBox::Show("Failed to set value");
else
    MessageBox::Show("Value changed successfully ");

Visual Basic

Dim result As TPOBDIIStatus
Dim iBuffer As UInt32 = 0
' Enable CAN DEBUG mode
iBuffer = OBDIIApi.POBDII_DEBUG_CAN
result = OBDIIApi.SetValue(OBDDIIApi.POBDII_PCIBUS2, 
    TOBDDIIParameter.POBDII_PARAM_DEBUG, iBuffer, Convert.ToUInt32(Len(iBuffer)))
If result <> TPOBDIIStatus.POBDII_ERROR_OK Then
    MessageBox.Show("Failed to set value")
Else
    MessageBox.Show("Value changed successfully ")
End If

Pascal OO

var
    result: TPOBDIIStatus;
    iBuffer: UINT;
begin
    // Enable CAN DEBUG mode
    iBuffer := TOBDIIapi.POBDII_DEBUG_CAN;
    result := TOBDIIapi.SetValue(TOBDIIapi.POBDII_PCIBUS2, POBDII_PARAM_DEBUG, PLongWord(@iBuffer), sizeof(iBuffer));
    if (result <> POBDII_ERROR_OK) then
        MessageBox(0, 'Failed to set value', 'Error', MB_OK)
    else
        MessageBox(0, 'Value changed successfully ', 'Error', MB_OK);
end;

See also: GetValue on page 55

Plain function version: OBDDII_GetValue
3.6.6 GetValue
Retrieves information from a POBDII Channel.

Overloads

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetValue(TPOBDIICANHandle, TPOBDIIParameter, StringBuilder, UInt32);</td>
<td>Retrieves information from a POBDII channel in text form</td>
</tr>
<tr>
<td>GetValue(TPOBDIICANHandle, TPOBDIIParameter, UInt32, UInt32);</td>
<td>Retrieves information from a POBDII channel in numeric form</td>
</tr>
<tr>
<td>GetValue(TPOBDIICANHandle, TPOBDIIParameter, Byte[], UInt32);</td>
<td>Retrieves information from a POBDII channel in a byte array form</td>
</tr>
</tbody>
</table>

3.6.7 GetValue (TPOBDIICANHandle, TPOBDIIParameter, StringBuilder, UInt32)
Retrieves information from a POBDII channel in text form.

Syntax
Pascal OO

```pascal
class function GetValue(  
    CanChannel: TPOBDIICANHandle;  
    Parameter: TPOBDIIParameter;  
    StringBuffer: PAnsiChar;  
    BufferLength: LongWord  
): TPOBDIIStatus; overload;
```

C#

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_GetValue")]
public static extern TPOBDIIStatus GetValue(  
    [MarshalAs(UnmanagedType.U2)]  
    TPOBDIICANHandle CanChannel,  
    [MarshalAs(UnmanagedType.U1)]  
    TPOBDIIParameter Parameter,  
    StringBuilder StringBuffer,  
    UInt32 BufferLength);  
```

C++ / CLR

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_GetValue")]
static TPOBDIIStatus GetValue(  
    [MarshalAs(UnmanagedType::U2)]  
    TPOBDIICANHandle CanChannel,  
    [MarshalAs(UnmanagedType::U1)]  
    TPOBDIIParameter Parameter,  
    StringBuilder^ StringBuffer,  
    UInt32 BufferLength);  
```

Visual Basic

```vb
<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_GetValue")>  
Public Shared Function GetValue(  
    ByVal CanChannel As TPOBDIICANHandle,  
    ...  
) As TPOBDIIStatus  
```

55
<MarshalAs(UnmanagedType.U1)> ByVal Parameter As TPOBDIIParameter, ByVal StringBuffer As StringBuilder, ByVal BufferLength As UInt32) As TPOBDIIStatus
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the value to be set (see TPOBDIIParameter on page 31)</td>
</tr>
<tr>
<td>StringBuffer</td>
<td>The buffer to return the required string value</td>
</tr>
<tr>
<td>BufferLength</td>
<td>The length in bytes of the given buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

| POBDII_ERROR_NOT_INITIALIZED | Indicates that the given POBDII channel cannot be uninitialized because it was not found in the list of reserved channels of the calling application |
| POBDII_ERROR_WRONG_PARAM    | Indicates that the parameters passed to the method are invalid. Check the value of 'Parameter' and assert it is compatible with an integer buffer |

Example

The following example shows the use of the method GetValue to retrieve the version of the OBDII API. Depending on the result, a message will be shown to the user.

C#

```csharp
TPOBDIIStatus result;
StringBuilder BufferString;

// Get API version
BufferString = new StringBuilder(255);
result = OBDIIApi.GetValue(OBDIIApi.POBDII_NONEBUS, TPOBDIIParameter.POBDII_PARAM_API_VERSION, BufferString, 255);
if (result != TPOBDIIStatus.POBDII_ERROR_OK)
    MessageBox.Show("Failed to get value");
else
    MessageBox.Show(BufferString.ToString());
```

C++ / CLR

```csharp
TPOBDIIStatus result;
StringBuilder^ BufferString;

// Get API version
BufferString = gcnew StringBuilder(255);
result = OBDIIApi::GetValue(OBDIIApi::POBDII_NONEBUS, POBDII_PARAM_API_VERSION, BufferString, 255);
if (result != POBDII_ERROR_OK)
    MessageBox::Show("Failed to get value");
else
    MessageBox::Show(BufferString->ToString());
```
Visual Basic

```vbnet
Dim result As TPOBDIStatus
Dim BufferString As StringBuilder

' Get API version
BufferString = New StringBuilder(255)
result = OBDIIApi.GetValue(OBDIIApi.POBDII_NONEBUS, 
    TPOBDIIParameter.POBDII_PARAM_API_VERSION, BufferString, 255)
If result <> TPOBDIIStatus.POBDII_ERROR_OK Then
    MessageBox.Show("Failed to get value")
Else
    MessageBox.Show(BufferString.ToString())
End If
```

Pascal OO

```pascal
var
    result: TPOBDIStatus;
    BufferString: array [0..256] of Char;

begin
    // Get API version
    result := TObdiiApi.GetValue(TObdiiApi.POBDII_NONEBUS, TPOBDII_PARAM_API_VERSION, BufferString, 255);
    if (result <> TPOBDII_ERROR_OK) then
        MessageBox(0, 'Failed to get value', 'Error', MB_OK)
    else
        MessageBox(0, BufferString, 'Success', MB_OK);
end;
```

See also: `SetValue` on page 52, `TPOBDIIParameter` on page 31, `Parameter Value Definitions` on page 124

Plain function version: `OBDII_GetValue`

### 3.6.8 GetValue (TPOBDIICANHandle, TPOBDIIParameter, UINT32, UINT32)

Retrieves information from a POBDII Channel in numeric form.

Syntax

**Pascal OO**

```pascal
class function GetValue(
    CanChannel: TPOBDIICANHandle;
    Parameter: TPOBDIIParameter;
    NumericBuffer: PLongWord;
    BufferLength: LongWord
): TPOBDIStatus; overload;
```

**C#**

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_GetValue")]
public static extern TPOBDIStatus GetValue(
    [MarshalAs(UnmanagedType.U2)]
    TPOBDIICANHandle CanChannel,
    [MarshalAs(UnmanagedType.U1)]
    TPOBDIIParameter Parameter,
    out UInt32 NumericBuffer,
    UInt32 BufferLength);
```
### C++ / CLR

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_GetValue")]
static TPOBDIIStatus GetValue(
    [MarshalAs(UnmanagedType.U2)]
    TPOBDIICANHandle CanChannel,
    [MarshalAs(UnmanagedType.U1)]
    TPOBDIIParameter Parameter,
    UInt32% NumericBuffer,
    UInt32 BufferLength);
```

### Visual Basic

```vbnet
<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_GetValue")> _
Public Shared Function GetValue( _
    ByVal CanChannel As TPOBDIICANHandle, _
    ByVal Parameter As TPOBDIIParameter, _
    ByRef NumericBuffer As UInt32, _
    ByVal BufferLength As UInt32) As TPOBDIIStatus
End Function
```

### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the value to be set (see TPOBDIIParameter on page 31)</td>
</tr>
<tr>
<td>NumericBuffer</td>
<td>The buffer to return the required numeric value</td>
</tr>
<tr>
<td>BufferLength</td>
<td>The length in bytes of the given buffer</td>
</tr>
</tbody>
</table>

### Returns

The return value is a [TPOBDIIStatus] code. **POBDII_ERROR_OK** is returned on success. The typical errors in case of failure are:

- **POBDII_ERROR_NOT_INITIALIZED**: Indicates that the given POBDII channel cannot be uninitialized because it was not found in the list of reserved channels of the calling application
- **POBDII_ERROR_WRONG_PARAM**: Indicates that the parameters passed to the method are invalid. Check the value of 'Parameter' and assert it is compatible with a string buffer

### Example

The following example shows the use of the method `GetValue` on the channel `POBDII_USBBUS1` to retrieve the number of connected and responding ECUs on the CAN bus. Depending on the result, a message will be shown to the user.

**Note:** It is assumed that the channel was already initialized.

**C#**

```csharp
TPCANTPHandle CanChannel = OBDIIApi.POBDII_USBBUS1;
TPOBDIIStatus result;
UInt32 iBuffer = 0;

// Get the number of connected ECUs
result = OBDIIApi.GetValue(CanChannel,
    TPOBDIIParameter.POBDII_PARAM_AVAILABLE_ECUS,
    out iBuffer, sizeof(UInt32));
if (result != TPOBDIIStatus.POBDII_ERROR_OK)
```
MessageBox.Show("Failed to get value");
else
    MessageBox.Show(iBuffer.ToString());

C++ / CLR

TPUDSCANHandle CanChannel = OBDIIApi::POBDII_USBBUS1;
TPOBDIIStatus result;
UInt32 iBuffer = 0;

// Get the number of connected ECUs
result = OBDIIApi::GetValue(CanChannel, POBDII_PARAM_AVAILABLE ECUS, iBuffer, sizeof(UInt32));
if (result != POBDII_ERROR_OK)
    MessageBox::Show("Failed to get value");
else
    MessageBox::Show(iBuffer.ToString());

Visual Basic

Dim result As TPOBDIIStatus
Dim BufferString As StringBuilder

' Get API version
BufferString = New StringBuilder(255)
result = OBDIIApi.GetValue(OBDIIApi.POBDII_NONEBUS, 
                           TPOBDIIParameter.POBDII_PARAM_API VERSION, BufferString, 255)
If result <> TPOBDIIStatus.POBDII_ERROR_OK Then
    MessageBox.Show("Failed to get value")
Else
    MessageBox.Show(BufferString.ToString())
End If

Pascal OO

var
  CanChannel: TPOBDIICANHandle;
  result: TPOBDIIStatus;
  iBuffer: UINT;

begin
  CanChannel := TObdiiApi.POBDII_USBBUS1;
  // Get the number of connected ECUs
  result := TObdiiApi.GetValue(CanChannel, POBDII_PARAM AVAILABLE ECUS, PLongWord(@iBuffer), sizeof(iBuffer));
  if (result <> POBDII_ERROR_OK) then
    MessageBox(0,'Failed to get value','Error',MB_OK)
  else
    MessageBox(0,PAnsiChar(AnsiString(Format('#ECU = %d', [Integer(iBuffer)]))),'Success',MB_OK);
end;

See also: SetValue on page 52, TPOBDIIParameter on page 31, Parameter Value Definitions on page 124.

Plain function version: OBDII_GetValue.
3.6.9 GetValue (TPOBDIIHandle, TPOBDIIParameter, Byte[], UInt32)

Retrieves information from a POBDII Channel in a byte array.

Syntax

C#  

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_GetValue")]
public static extern TPOBDIIStatus GetValue(
   [MarshalAs(UnmanagedType.U2)]
   TPOBDIICANHandle CanChannel,
   [MarshalAs(UnmanagedType.U1)]
   TPOBDIIParameter Parameter,
   [MarshalAs(UnmanagedType.LPArray)]
   [Out] Byte[] Buffer,
   UInt32 BufferLength);
```

C++ / CLR

```csharp
[DllImport("PCAN-OBDII.dll",EntryPoint="OBDII_GetValue")]
static TPOBDIIStatus GetValue(
   [MarshalAs(UnmanagedType::U2)]
   TPOBDIICANHandle CanChannel,
   [MarshalAs(UnmanagedType::U1)]
   TPOBDIIParameter Parameter,
   [MarshalAs(UnmanagedType::LPArray)]
   array<Byte>^ Buffer,
   UInt32 BufferLength);
```

Visual Basic

```vbnet
<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_GetValue")> _
Public Shared Function GetValue( _
   ByVal CanChannel As TPOBDIICANHandle, _
   ByVal Parameter As TPOBDIIParameter, _
   ByVal Buffer As Byte(), _
   ByVal BufferLength As UInt32) As TPOBDIIStatus
End Function
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the value to be set (see TPOBDIIParameter on page 31)</td>
</tr>
<tr>
<td>Buffer</td>
<td>The buffer containing the array value to retrieve</td>
</tr>
<tr>
<td>BufferLength</td>
<td>The length in bytes of the given buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>POBDII_ERROR_NOT_INITIALIZED</th>
<th>Indicates that the given POBDII channel cannot be uninitialized because it was not found in the list of reserved channels of the calling application</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_ERROR_WRONG_PARAM</td>
<td>Indicates that the parameters passed to the method are invalid. Check the value of 'Parameter' and assert it is compatible with a string buffer</td>
</tr>
</tbody>
</table>
Example
The following example shows the use of the method `GetValue` on the channel `POBDII_USBBUS1` to retrieve the supported Parameter Identifiers (PIDs) by the connected ECUs. Depending on the result, a message will be shown to the user.

**Note:** It is assumed that the channel was already initialized.

C#  
```csharp
tpcantphandle canchannel = obdiiapi.pobdi_usbbus1;
TPOBDIStatus result;
uint bufferLength = 256;
byte[] bufferArray = new byte[bufferLength];

// Get the supported PIDs list
result = OBDIIApi.GetValue(canchannel, TPOBDIParameter.POBDII_PARAM_SUPPORTMASK_PIDS, bufferArray, sizeof(byte) * bufferLength);
if (result != TPOBDIStatus.POBDII_ERROR_OK)
    MessageBox.Show("Failed to get value");
else
    MessageBox.Show("Successfully retrieved supported PIDs");
```

C++ / CLR  
```csharp
TPUDSCANHandle canchannel = OBDIIApi::POBDII_USBBUS1;
TPOBDIStatus result;
UInt32 bufferLength = 256;
array<Byte>* bufferArray = gcnew array<Byte>(bufferLength);

// Get the supported PIDs list
result = OBDIIApi::GetValue(canchannel, POBDII_PARAM_SUPPORTMASK_PIDS, bufferArray, sizeof(Byte) * bufferLength);
if (result != POBDII_ERROR_OK)
    MessageBox::Show("Failed to get value");
else
    MessageBox::Show("Successfully retrieved supported PIDs");
```

Visual Basic  
```vb
Dim canchannel As TPCANTPHandle = OBDIIApi.POBDII_USBBUS1
Dim result As TPOBDIStatus
Dim bufferLength As UInt32 = 256
Dim bufferArray(bufferLength) As Byte

' Get the supported PIDs list
result = OBDIIApi.GetValue(canchannel, TPOBDIParameter.POBDII_PARAM_SUPPORTMASK_PIDS, bufferArray, Convert.ToUInt32(bufferArray.Length))
If result <> TPOBDIStatus.P0BDII_ERROR_OK Then
    MessageBox.Show("Failed to get value")
Else
    MessageBox.Show("Successfully retrieved supported PIDs")
End If
```

Pascal OO  
```pascal
var
    canchannel: TPOBDIICANHandle;
```
result: TPOBDIIStatus;
bufferArray: array [0..255] of Byte;

begin
CanChannel := TObdiiApi.POBDII_USBUSB1;

// Get the Supported Parameter IDs
result := TObdiiApi.GetValue(CanChannel, POBDII_PARAM_SUPPORTMASK_PIDS, PLongWord(@bufferArray), Length(bufferArray));
if (result <> POBDII_ERROR_OK) then
  MessageBox(0, 'Failed to get value', 'Error', MB_OK)
else
  MessageBox(0, 'Successfully retrieved supported PIDs', 'Success', MB_OK)
end;

See also: SetValue on page 52, TPOBDIIParameter on page 31, Parameter Value Definitions on page 124

Plain function version: OBDII_GetValue

3.6.10 GetStatus
Gets the current BUS status of a POBDII Channel.

Syntax
Pascal OO

class function GetStatus(  
  CanChannel: TPOBDIICANHandle  
): TPOBDIIStatus;

C#

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_GetStatus")]
public static extern TPOBDIIStatus GetStatus(  
  [MarshalAs(UnmanagedType.U2)]  
  TPOBDIICANHandle CanChannel);

C++ / CLR

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_GetStatus")]
static TPOBDIIStatus GetStatus(  
  [MarshalAs( UnmanagedType::U2)]  
  TPOBDIICANHandle CanChannel);

Visual Basic

<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_GetStatus")> _  
Public Shared Function GetStatus( _  
  ByVal CanChannel As TPOBDIICANHandle) As TPOBDIIStatus  
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
</tbody>
</table>
Returns
The return value is a `TPOBDIIStatus` code. `POBDII_ERROR_OK` is returned on success. The typical errors in case of failure are:

- **POBDII_ERROR_OK**: Indicates that the status of the given POBDII channel is OK
- **POBDII_ERROR_BUSLIGHT**: Indicates a bus error within the given POBDII Channel. The hardware is in bus-light status
- **POBDII_ERROR_BUSHEAVY**: Indicates a bus error within the given POBDII Channel. The hardware is in bus-heavy status
- **POBDII_ERROR_BUSOFF**: Indicates a bus error within the given POBDII Channel. The hardware is in bus-off status
- **POBDII_ERROR_NOT_INITIALIZED**: Indicates that the given POBDII channel cannot be used because it was not found in the list of reserved channels of the calling application

Remarks: When the hardware status is bus-off, an application cannot communicate anymore. Consider using the PCAN-Basic property `PCAN_BUSOFF_AUTORESET` which instructs the API to automatically reset the CAN controller when a bus-off state is detected.

Another way to reset errors like bus-off, bus-heavy and bus-light, is to uninitialize and initialize again the channel used. This causes a hardware reset.

Example
The following example shows the use of the method `GetStatus` on the channel `POBDII_PCIBUS1`. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```csharp
TPOBDIIStatus result;

// Check the status of the PCI Channel
result = OBDIIApi.GetStatus(OBDIIApi.POBDII_PCIBUS1);
switch (result)
{
    case TPOBDIIStatus.POBDII_ERROR_BUSLIGHT:
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-LIGHT status...");
        break;
    case TPOBDIIStatus.POBDII_ERROR_BUSHEAVY:
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-HEAVY status...");
        break;
    case TPOBDIIStatus.POBDII_ERROR_BUSOFF:
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-OFF status...");
        break;
    case TPOBDIIStatus.POBDII_ERROR_OK:
        MessageBox.Show("PCAN-PCI (Ch-1): Status is OK");
        break;
    default:
        // An error occurred;
        MessageBox.Show("Failed to retrieve status");
        break;
}
```

C++ / CLR

```csharp
TPOBDIIStatus result;

// Check the status of the PCI Channel
result = OBDIIApi::GetStatus(OBDIIApi::POBDII_PCIBUS1);
```
switch (result)
{
case POBDII_ERROR_BUSLIGHT:
    MessageBox::Show("PCAN-PCI (Ch-1): Handling a BUS-LIGHT status...");
    break;
case POBDII_ERROR_BUSHEAVY:
    MessageBox::Show("PCAN-PCI (Ch-1): Handling a BUS-HEAVY status...");
    break;
case POBDII_ERROR_BUSOFF:
    MessageBox::Show("PCAN-PCI (Ch-1): Handling a BUS-OFF status...");
    break;
case POBDII_ERROR_OK:
    MessageBox::Show("PCAN-PCI (Ch-1): Status is OK");
    break;
default:
    // An error occurred
    MessageBox::Show("Failed to retrieve status");
    break;
}

Visual Basic

Dim result As TPOBDIIStatus

' Check the status of the PCI Channel
result = OBDIIApi.GetStatus(OBDIIApi.POBDII_PCIBUS1)
Select Case (result)
    Case TPOBDIIStatus.POBDII_ERROR_BUSLIGHT
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-LIGHT status...")
    Case TPOBDIIStatus.POBDII_ERROR_BUSHEAVY
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-HEAVY status...")
    Case TPOBDIIStatus.POBDII_ERROR_BUSOFF
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-OFF status...")
    Case TPOBDIIStatus.POBDII_ERROR_OK
        MessageBox.Show("PCAN-PCI (Ch-1): Status is OK")
    Case Else
        ' An error occurred
        MessageBox.Show("Failed to retrieve status")
End Select
Pascal OO

var
    result: TPOBDIIStatus;

begin
    // Check the status of the PCI Channel
    result := TObdiiApi.GetStatus(TObdiiApi.POBDII_PClBUS1);
    Case (result) of
        POBDII_ERROR_BUSLIGHT:
            MessageBox(0, 'PCAN-PCI (Ch-1): Handling a BUS-LIGHT status...', 'Error', MB_OK);
        POBDII_ERROR_BUSHEAVY:
            MessageBox(0, 'PCAN-PCI (Ch-1): Handling a BUS-HEAVY status...', 'Error', MB_OK);
        POBDII_ERROR_BUSOFF:
            MessageBox(0, 'PCAN-PCI (Ch-1): Handling a BUS-OFF status...', 'Error', MB_OK);
        POBDII_ERROR_OK:
            MessageBox(0, 'PCAN-PCI (Ch-1): Status is OK', 'Error', MB_OK);
        else
            // An error occurred
            MessageBox(0, 'Failed to retrieve status', 'Error', MB_OK);
    end;
end;

Plain function version: OBDII_GetStatus

3.6.11 Reset

Resets the receive and transmit queues of a POBDII Channel.

Syntax

Pascal OO

class function Reset(
    CanChannel: TPOBDIICANHandle
): TPOBDIIStatus;

C#

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_Reset")]
public static extern TPOBDIIStatus Reset(
    [MarshalAs(UnmanagedType.U2)]
    TPOBDIICANHandle CanChannel);

C++ / CLR

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_Reset")]
static TPOBDIIStatus Reset(
    [MarshalAs(UnmanagedType::U2)]
    TPOBDIICANHandle CanChannel);
Visual Basic

```vbnet
<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_Reset")>  
Public Shared Function Reset(  
    ByVal CanChannel As TPOBDIICANHandle) As TPOBDIIStatus
End Function
```

### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
</tbody>
</table>

### Returns

The return value is a [TPOBDIIStatus] code. **POBDII_ERROR_OK** is returned on success. The typical errors in case of failure are:

- **POBDII_ERROR_NOT_INITIALIZED**: Indicates that the given POBDII channel cannot be used because it was not found in the list of reserved channels of the calling application.

### Remarks

Calling this method ONLY clears the queues of a Channel. A reset of the CAN controller doesn’t take place.

### Example

The following example shows the use of the method Reset on the channel **POBDII_PCIBUS1**. Depending on the result, a message will be shown to the user.

> **Note**: It is assumed that the channel was already initialized.

### C#

```csharp
TPOBDIIStatus result;

// The PCI Channel is reset
result = OBDIIApi.Reset(OBDIIApi.POBDII_PCIBUS1);
if (result != TPOBDIIStatus.POBDII_ERROR_OK)
{
    // An error occurred
    MessageBox.Show("An error occurred");
}
else
    MessageBox.Show("PCAN-PCI (Ch-1) was reset");
```

### C++/CLR

```csharp
TPOBDIIStatus result;

// The PCI Channel is reset
result = OBDIIApi::Reset(OBDIIApi::POBDII_PCIBUS1);
if (result != POBDII_ERROR_OK)
{
    // An error occurred
    MessageBox::Show("An error occurred");
}
else
    MessageBox::Show("PCAN-PCI (Ch-1) was reset");
```
Visual Basic

Dim result As TPOBDIIStatus

' The PCI Channel is reset
result = OBDIIApi.Reset(OBDIIApi.POBDII_PCIBUS1)
If result <> TPOBDIIStatus.POBDII_ERROR_OK Then
   ' An error occurred
   MessageBox.Show("An error occurred")
Else
   MessageBox.Show("PCAN-PCI (Ch-1) was reset")
End If

Pascal OO

var
  result: TPOBDIIStatus;

begin
  // The PCI Channel is reset
  result := TObdiiApi.Reset(TObdiiApi.POBDII_PCIBUS1);
  if (result <> POBDII_ERROR_OK) then
     // An error occurred
     MessageBox(0, 'An error occurred', 'Error', MB_OK)
  else
     MessageBox(0, 'PCAN-PCI (Ch-1) was reset', 'Error', MB_OK);
end;

See also: Uninitialize on page 50

Plain function version: OBDII_Reset

3.6.12 GetUnitAndScaling

Retrieves information from a Unit and Scaling Identifier.

Syntax

Pascal OO

class function GetUnitAndScaling(
  id: Byte;
  unitAndScaling: PTPOBDIIUnitAndScaling): TPOBDIIStatus; stdcall;

C#

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_GetUnitAndScaling")]
public static extern TPOBDIIStatus GetUnitAndScaling(
  Byte id,
  out TPOBDIIUnitAndScaling unitAndScaling);

C++ / CLR

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_GetUnitAndScaling")]
static TPOBDIIStatus GetUnitAndScaling(
  Byte id,
  [Out] TPOBDIIUnitAndScaling *unitAndScaling);
**Visual Basic**

```vbnet
<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_GetUnitAndScaling")> _
Public Shared Function GetUnitAndScaling( _
    ByVal id As Byte, _
    ByRef unitAndScaling As TPOBDIIUnitAndScaling) As TPOBDIIStatus
End Function
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>The Unit and Scaling Identifier to retrieve information from</td>
</tr>
<tr>
<td>UnitAndScanling</td>
<td>A buffer to store the TPOBDIIUnitAndScaling information</td>
</tr>
</tbody>
</table>

**Returns**

The return value is a [TPOBDIIStatus] code. [POBDII_ERROR_OK] is returned on success. The typical errors in case of failure are:

- [POBDII_ERROR_NOT_INITIALIZED] Indicates that the given POBDII channel cannot be used because it was not found in the list of reserved channels of the calling application.

**Example**

The following example shows the use of the method `GetUnitAndScaling` with the Voltage ID (0x0A). Depending on the result, a message will be shown to the user.

**Note:** It is assumed that the channel was already initialized.

**C#**

```csharp
TPOBDIIStatus result;
TPOBDIIUnitAndScaling unitAndScaling;

// Get Unit and Scaling information
result = OBDIIApi.GetUnitAndScaling(0x0A, out unitAndScaling);
if (result != TPOBDIIStatus.POBDII_ERROR_OK)
{
    // An error occurred
    MessageBox.Show("An error occurred");
}
else
    MessageBox.Show("Unit: " + unitAndScaling.UNIT);
```

**C++/CLR**

```cpp
TPOBDIIStatus result;
TPOBDIIUnitAndScaling unitAndScaling;

// Get Unit and Scaling information
result = OBDIIApi::GetUnitAndScaling(0x0A, unitAndScaling);
if (result != POBDII_ERROR_OK)
{
    // An error occurred
    MessageBox::Show("An error occurred");
}
else
    MessageBox::Show("Unit: " + unitAndScaling.UNIT);
```
### Visual Basic

```vbnet
Dim result As TPOBDIIStatus
Dim unitAndScaling As TPOBDIIUnitAndScaling

' Get Unit and Scaling information
result = OBDIIApi.GetUnitAndScaling(&HA, unitAndScaling)
If (result <> TPOBDIIStatus.POBDII_ERROR_OK) Then
  ' An error occurred
  MessageBox.Show("An error occurred")
Else
  MessageBox.Show(String.Format("Unit: {0}", unitAndScaling.UNIT))
End If
```

### Pascal OO

```pascal
var
  result: TPOBDIIStatus;
  unitAndScaling: TPOBDIIUnitAndScaling;
begin
  // The PCI Channel is reset
  result := TObdiiApi.GetUnitAndScaling($0A, @unitAndScaling);
  if (result <> POBDII_ERROR_OK) then
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK)
  else
    MessageBox(0, PAnsiChar(AnsiString(Format('Unit = %s', [unitAndScaling.UNIT_TXT])))), 'Success', MB_OK);
end;
```

See also: [RequestTestResults](#) on page 83

Plain function version: [OBDII_GetUnitAndScaling](#)

### 3.6.13 RequestCurrentData

Sends an OBDII Service “Request Current Powertrain Diagnostic Data” ($01) request into queue and waits to receive the responses. The purpose of this service is to allow access to current emission-related data values, including analogue inputs and outputs, digital inputs and outputs, and system status information.

**Syntax**

**Pascal OO**

```pascal
class function RequestCurrentData(
  CanChannel: TPOBDIICANHandle;
  Pid: TPOBDIIPid;
  Data: PTPOBDIIParamData;
  DataLen: Byte
): TPOBDIIStatus;
```

**C#**

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestCurrentData")
public static extern TPOBDIIStatus RequestCurrentData(  
  [MarshalAs(UnmanagedType.U1)]  
  TPOBDIICANHandle CanChannel,  
  [MarshalAs(UnmanagedType.U1)]  
  Byte DataLen: Byte  
): TPOBDIIStatus;
```
TPOBDIIPid Pid,
    [In, Out]
    TPOBDIIParamData[] Data,
    byte DataLen);

C++ / CLR

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestCurrentData")]
static TPOBDIIStatus RequestCurrentData(
    [MarshalAs(UnmanagedType::U1)]
    TPOBDIICANHandle CanChannel,
    [MarshalAs(UnmanagedType::U1)]
    TPOBDIIPid Pid,
    [Out] array<TPOBDIIParamData>^ Data,
    Byte DataLen);

Visual Basic

<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_RequestCurrentData")> _
Public Shared Function RequestCurrentData( _
    <MarshalAs(UnmanagedType.U1)> _
    ByVal CanChannel As TPOBDIICANHandle, _
    <MarshalAs(UnmanagedType.U1)> _
    ByVal Pid As TPOBDIIPid, _
    <[In](), Out()> ByVal Data As TPOBDIIParamData(), _
    ByVal DataLen As Byte) As TPOBDIIStatus
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Pid</td>
<td>The Parameter Identifier to request</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIParamData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>TPOBDIIStatus</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_ERROR_NOT_INITIALIZED</td>
<td>Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MESSAGE</td>
<td>Indicates that no matching message was received</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MEMORY</td>
<td>Failed to allocate a buffer to store the expected responses</td>
</tr>
</tbody>
</table>

Remarks: User should always check the error code of the generic response to assert that the response data is valid.

Example

The following example shows the use of the method RequestCurrentData on the channel POBDII_USB0 with PID 01. Depending on the result, a message will be shown to the user. If responses are received, a loop is set to handle valid responses.

Note: It is assumed that the channel was already initialized.
C#

TPOBDIStatus result;
byte bufferLength = 8;
TPOBDIIParamData[] buffer = new TPOBDIIParamData[8];

// Send OBDII Service $01 request
result = OBDIIApi.RequestCurrentData(OBDIIApi.POBDII_USBBUS1, 0x01, buffer, bufferLength);
if (result == TPOBDIStatus.POBDII_ERROR_OK)
{
    MessageBox.Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == TPOBDIIError.POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == TPOBDIStatus.POBDII_ERROR_NO_MESSAGE)
    MessageBox.Show("Request received no response.");
else
    MessageBox.Show("An error occurred.");

C++/CLR

TPOBDIStatus result;
Byte bufferLength = 8;
array<TPOBDIIParamData>^ buffer = gcnew array<TPOBDIIParamData>(8);

// Send OBDII Service $01 request
result = OBDIIApi::RequestCurrentData(OBDIIApi::POBDII_USBBUS1, 0x01, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox::Show("Request received responses.");
    // loop through responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox::Show("Request received no response.");
else
// An error occurred
MessageBox::Show("An error occurred.");

Visual Basic

Dim result As TPOBDIIStatus
Dim bufferLength As Byte = 8
Dim buffer(bufferLength) As TPOBDIIParamData

' Send OBDII Service $01 request
result = OBDIIApi.RequestCurrentData(OBDIIApi.POBDII_USBBUS1, &H1, buffer, bufferLength)
If (result = TPOBDIIStatus.POBDII_ERROR_OK) Then
    MessageBox.Show("Request received responses.")
    ' search and remove unused responses
    For j As Integer = 0 To bufferLength - 1
        If (buffer(j).RESPONSE.ERRORNR = TPOBDIIError.POBDII_R_NOT_USED) Then
            ' unused response, stop
            Exit For
        Else
            ' process response
        End If
    Next
ElseIf (result = TPOBDIIStatus.POBDII_ERROR_NO_MESSAGE) Then
    MessageBox.Show("Request received no response.")
Else
    ' An error occurred
    MessageBox.Show("An error occurred.")
End If

Pascal OO

var
    result: TPOBDIIStatus;
    bufferLength: byte;
    buffer: array [0..7] of TPOBDIIParamData;
    j: integer;
begin
    bufferLength := 8;
    // Send OBDII Service $01 request
    result := TObdiiApi.RequestCurrentData(TObdiiApi.POBDII_USBBUS1, $01, @buffer, bufferLength);
    if (result = POBDII_ERROR_OK) then
        begin
            MessageBox(0, 'Request received responses.', 'Success', MB_OK);
            // search and remove unused responses
            for j := 0 To bufferLength - 1 do
                begin
                    if (buffer[j].RESPONSE.ERRORNR = POBDII_R_NOT_USED) then
                        // unused response, stop
                        break
                    else
                        // process response
                    end
                end
        end
    end

if (result = POBDII_ERROR_NO_MESSAGE) then
    MessageBox(0, 'Request received no response.', 'Success', MB_OK)
else
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK)
end;

See also: TPOBDIIParamData on page 14

Plain function version: OBDII_RequestCurrentData

3.6.14 RequestFreezeFrameData

Sends an OBDII Service “Request Powertrain Freeze Frame Data” ($02) request into queue and waits to receive the responses. The purpose of this service is to allow access to emission-related data values in a freeze frame.

Syntax

Pascal OO

class function RequestFreezeFrameData(  
    CanChannel: TPOBDIICANHandle;  
    Pid: TPOBDIIPid;  
    Frame: Byte;  
    Data: PTPOBDIIParamData;  
    DataLen: Byte  
): TPOBDIIStatus;

C#  

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestFreezeFrameData")]
public static extern TPOBDIIStatus RequestFreezeFrameData(  
    [MarshalAs(UnmanagedType.U1)]  
    TPOBDIICANHandle CanChannel,  
    [MarshalAs(UnmanagedType.U1)]  
    TPOBDIIPid Pid,  
    byte Frame,  
    [In, Out]  
    TPOBDIIParamData[] Data,  
    byte DataLen);

C++ / CLR

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestFreezeFrameData")]
static TPOBDIIStatus RequestFreezeFrameData(  
    [MarshalAs(UnmanagedType::U1)]  
    TPOBDIICANHandle CanChannel,  
    [MarshalAs(UnmanagedType::U1)]  
    TPOBDIIPid Pid,  
    Byte Frame,  
    [Out] array<TPOBDIIParamData>^ Data,  
    Byte DataLen);

Visual Basic

<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_RequestFreezeFrameData")> _  
Public Shared Function RequestFreezeFrameData( _  
    <MarshalAs(UnmanagedType.U1)> _
ByVal CanChannel As TPOBDIICANHandle, _
<MarshalAs(UnmanagedType.U1)> ByVal Pid As TPOBDIIPid, _
ByVal Frame As Byte,
<[In](), Out()> ByVal Data As TPOBDIIParamData(), _
ByVal DataLen As Byte) As TPOBDIIStatus
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Pid</td>
<td>The Parameter Identifier to request</td>
</tr>
<tr>
<td>Frame</td>
<td>The Freeze Frame number</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIParamData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

- **POBDII_ERROR_NOT_INITIALIZED**: Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application
- **POBDII_ERROR_NO_MESSAGE**: Indicates that no matching message was received
- **POBDII_ERROR_NO_MEMORY**: Failed to allocate a buffer to store the expected responses

Remarks: User should always check the error code of the generic response to assert that the response data is valid.

Example

The following example shows the use of the method `RequestFreezeFrameData` on the channel `POBDII_USBBUS1` with PID 02 and Freeze Frame 00 (i.e. read the DTC that caused the freeze frame data to be stored).

Depending on the result, a message will be shown to the user.

**Note:** It is assumed that the channel was already initialized.

C#

```csharp
TPOBDIIStatus result;
byte bufferLength = 8;
TPOBDIIParamData[] buffer = new TPOBDIIParamData[8];

// Send OBDII Service $02 request (PID 02 and with frame 00)
result = OBDIIApi.RequestFreezeFrameData(OBDIIApi.POBDII_USBBUS1, 0x02, 0x00, buffer, bufferLength);
if (result == TPOBDIIStatus.POBDII_ERROR_OK)
{
    MessageBox.Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == TPOBDIIError.POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
```
// process response

else if (result == TPOBDIIStatus.POBDII_ERROR_NO_MESSAGE)
    MessageBox.Show("Request received no response.");
else
    // An error occurred
    MessageBox.Show("An error occurred.");

C++/CLR

TPOBDIIStatus result;
Byte bufferLength = 8;
array<TPOBDIIParamData>^ buffer = gcnew array<TPOBDIIParamData>(8);

// Send OBDII Service $02 request (PID 02 and with frame 00)
result = OBDIIApi::RequestFreezeFrameData(OBDIIApi::POBDII_USBBUS1, 0x02, 0x00, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox::Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }

else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox::Show("Request received no response.");
else
    // An error occurred
    MessageBox::Show("An error occurred.");

Visual Basic

Dim result As TPOBDIIStatus
Dim bufferLength As Byte = 8
Dim buffer(bufferLength) As TPOBDIIParamData

' Send OBDII Service $02 request (PID 02 and with frame 00)
result = OBDIIApi.RequestFreezeFrameData(OBDIIApi.POBDDI_USBBUS1, &H02, &H00, buffer, bufferLength)
If (result = TPOBDIIStatus.POBDII_ERROR_OK) Then
    MessageBox.Show("Request received responses.")
    ' search and remove unused responses
    For j As Integer = 0 To bufferLength - 1
        If (buffer(j).RESPONSE.ERRORNR = POBDIIRerror.POBDDI_R_NOT_USED) Then
            ' unused response, stop
            Exit For
        Else
            ' process response
    End If
    Else
        ' An error occurred
        MessageBox.Show("An error occurred.");
    End If
Next
ElseIf (result = TPOBDIIStatus.POBDII_ERROR_NO_MESSAGE) Then
    MessageBox.Show("Request received no response.")
Else
    ' An error occurred
    MessageBox.Show("An error occurred.")
End If

Pascal OO

var
    result: TPOBDIIStatus;
    bufferLength: byte;
    buffer: array [0..7] of TPOBDIIParamData;
    j: integer;

begin
    bufferLength := 8;

    // Send OBDII Service $02 request (PID 02 and with frame 00)
    result := TObdiiApi.RequestFreezeFrameData(TObdiiApi.POBDII_USBBUS1, $02, $00, @buffer, bufferLength);
    if (result = POBDII_ERROR_OK) then
        begin
            MessageBox(0, 'Request received responses.', 'Success', MB_OK);
            // search and remove unused responses
            for j := 0 To bufferLength - 1 do
                begin
                    if (buffer[j].RESPONSE.ERRORNR = POBDII_R_NOT_USED) then
                        // unused response, stop
                        break
                    else
                        // process response
                        ;
                end
        end
    else
        if (result = POBDII_ERROR_NO_MESSAGE) then
            MessageBox(0, 'Request received no response.', 'Success', MB_OK)
        else
            // An error occurred
            MessageBox(0, 'An error occurred', 'Error', MB_OK)
    end;

See also: TPOBDIIParamData on page 14

Plain function version: OBDII_RequestFreezeFrameData

3.6.15 RequestStoredTroubleCodes

Sends an OBDII Service “Request Emission-Related Diagnostic Information” ($03) request into queue and waits to receive the responses. The purpose of this service is to enable the external test equipment to obtain “confirmed” emission-related DTCs.
Syntax

Pascal OO

```pascal
class function RequestStoredTroubleCodes(
  CanChannel: TPOBDIICANHandle;
  Data: PTPOBDIIDTCData;
  DataLen: Byte
): TPOBDIIStatus;
```

C#

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestStoredTroubleCodes")]
public static extern TPOBDIIStatus RequestStoredTroubleCodes(
  [MarshalAs(UnmanagedType.U1)]
  TPOBDIICANHandle CanChannel,
  [In, Out]
  TPOBDIIDTCData[] Data,
  byte DataLen);
```

C++ / CLR

```cpp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestStoredTroubleCodes")]
static TPOBDIIStatus RequestStoredTroubleCodes(
  [MarshalAs(UnmanagedType::U1)]
  TPOBDIICANHandle CanChannel,
  [Out] array<TPOBDIIDTCData>^ Data,
  Byte DataLen);
```

Visual Basic

```vbnet
<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_RequestStoredTroubleCodes")> _
Public Shared Function RequestStoredTroubleCodes( _
  ByVal CanChannel As TPOBDIICANHandle, _
  ByRef Data As TPOBDIIDTCData(), _
  ByVal DataLen As Byte) As TPOBDIIStatus
End Function
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIDTCData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. `TPOBDII_ERROR_OK` is returned on success. The typical errors in case of failure are:

- `TPOBDII_ERROR_NOT_INITIALIZED` Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application
- `TPOBDII_ERROR_NO_MESSAGE` Indicates that no matching message was received
- `TPOBDII_ERROR_NO_MEMORY` Failed to allocate a buffer to store the expected responses

Remarks: User should always check the error code of the generic response to assert that the response data is valid.
Example

The following example shows the use of the method `RequestStoredTroubleCodes` on the channel `POBDII_USBBUS1`. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```csharp
TPOBDIIStatus result;
byte bufferLength = 8;
TPOBDIIDTCData[] buffer = new TPOBDIIDTCData[8];

// Send OBDII Service 03 request
result = OBDIIApi.RequestStoredTroubleCodes(OBDIIApi.POBDII_USBBUS1, buffer, bufferLength);
if (result == TPOBDIIStatus.POBDII_ERROR_OK)
{
    MessageBox.Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == TPOBDIIError.POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == TPOBDIIStatus.POBDII_ERROR_NO_MESSAGE)
    MessageBox.Show("Request received no response.");
else
    // An error occurred
    MessageBox.Show("An error occurred.");
```

C++/CLR

```cpp
TPOBDIIStatus result;
Byte bufferLength = 8;
array<TPOBDIIDTCData>^ buffer = gcnew array<TPOBDIIDTCData>(8);

// Send OBDII Service 03 request
result = OBDIIApi::RequestStoredTroubleCodes(OBDIIApi::POBDII_USBBUS1, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox::Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
        }
    }
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox::Show("Request received no response.");
else
    // An error occurred
    MessageBox::Show("An error occurred.");
```
// process response

else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox::Show("Request received no response.");
else
    // An error occurred
    MessageBox::Show("An error occurred.");

Visual Basic

Dim result As TPOBDIIStatus
Dim bufferLength As Byte = 8
Dim buffer(bufferLength) As TPOBDIIDTCData

' Send OBDII Service $03 request
result = OBDIIApi.RequestStoredTroubleCodes(OBDIIApi.POBDII_USBBUS1, buffer, bufferLength)
If (result = TPOBDIIStatus.POBDII_ERROR_OK) Then
    MessageBox.Show("Request received responses.")
    ' search and remove unused responses
    For j As Integer = 0 To bufferLength - 1
        If (buffer(j).RESPONSE.ERRORNR = TPOBDIIError.POBDII_R_NOT_USED) Then
            ' unused response, stop
            Exit For
        Else
            ' process response
        End If
    Next
ElseIf (result = TPOBDIIStatus.POBDII_ERROR_NO_MESSAGE) Then
    MessageBox.Show("Request received no response.")
Else
    ' An error occurred
    MessageBox.Show("An error occurred.")
End If

Pascal OO

var
    result: TPOBDIIStatus;
    bufferLength: byte;
    buffer: array [0..7] of TPOBDIIDTCData;
    j: integer;

begin
    bufferLength := 8;

    // Send OBDII Service $03 request
    result := TObdiiApi.RequestStoredTroubleCodes(TObdiiApi.POBDII_USBBUS1, @buffer, bufferLength);
    if (result = POBDII_ERROR_OK) then
        begin
            MessageBox(0, 'Request received responses.', 'Success', MB_OK);
            ' search and remove unused responses
            for j := 0 To bufferLength - 1 do
                begin
                    if (buffer[j].RESPONSE.ERRORNR = POBDII_R_NOT_USED) then
                        "An error occurred";
                    End If
                End For
        End If
    Else
        "An error occurred";
    End If
// unused response, stop
break
else
    // process response
;
end
end
else if (result = POBDII_ERROR_NO_MESSAGE) then
    MessageBox(0, 'Request received no response.', 'Success', MB_OK)
else
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK)
end;

See also: TPOBDIIDTCData on page 17

Plain function version: OBDII_RequestStoredTroubleCodes

3.6.16 ClearTroubleCodes

Sends an OBDII Service “Clear/Reset Emission-Related Diagnostic Information” ($04) request into queue and waits to receive the responses. The purpose of this service is to provide a means for the external test equipment to command ECUs to clear all emission-related diagnostic information.

Syntax

Pascal OO

class function ClearTroubleCodes(
    CanChannel: TPOBDIICANHandle;
    Data: PTPOBDIIResponse;
    DataLen: Byte
): TPOBDIIStatus;

C#  

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_ClearTroubleCodes")]
public static extern TPOBDIIStatus ClearTroubleCodes(
    [MarshalAs(UnmanagedType.U1)]
    TPOBDIICANHandle CanChannel,
    [In, Out]
    TPOBDIIResponse[] Data,
    byte DataLen);

C++ / CLR

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_ClearTroubleCodes")]
static TPOBDIIStatus ClearTroubleCodes(
    [MarshalAs(UnmanagedType::U1)]
    TPOBDIICANHandle CanChannel,
    [Out] array<TPOBDIIResponse>^ Data,
    Byte DataLen);

Visual Basic

<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_ClearTroubleCodes")>
Public Shared Function ClearTroubleCodes( _
<MarshalAs(UnmanagedType.U1)> ByVal CanChannel As TPOBDIICANHandle, _
<[In](), Out()> ByVal Data As TPOBDIIResponse(), _
ByVal DataLen As Byte) As TPOBDIStatus
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIResponse structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

- POBDII_ERROR_NOT_INITIALIZED Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application
- POBDII_ERROR_NO_MESSAGE Indicates that no matching message was received
- POBDII_ERROR_NO_MEMORY Failed to allocate a buffer to store the expected responses

Remarks: User should always check the error code of the generic response to assert that the response data is valid.

Example

The following example shows the use of the method ClearTroubleCodes on the channel POBDII_USBBUS1. Depending on the result, a message will be shown to the user.

> Note: It is assumed that the channel was already initialized.

C#

```csharp
TPOBDIStatus result;
byte bufferLength = 8;
TPOBDIIResponse[] buffer = new TPOBDIIResponse[8];

// Send OBDII Service $04 request
result = OBDIIApi.ClearTroubleCodes(OBDIIApi.POBDII_USBBUS1, buffer, bufferLength);
if (result == TPOBDIStatus.POBDII_ERROR_OK)
{
    MessageBox.Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].ERRORNR == TPOBDIIError.POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == TPOBDIStatus.POBDII_ERROR_NO_MESSAGE)
    MessageBox.Show("Request received no response.");
else
```
```csharp
// An error occurred
MessageBox.Show("An error occured.");

C++/CLR

TPOBDIIStatus result;
Byte bufferLength = 8;
array<TPOBDIIResponse>^ buffer = gcnew array<TPOBDIIResponse>(8);

// Send OBDII Service $04 request
result = OBDIIApi::ClearTroubleCodes(OBDIIApi::POBDII_USBBUS1, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{

    MessageBox::Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox::Show("Request received no response.");
else

    // An error occurred
    MessageBox::Show("An error occured.");

Visual Basic

Dim result As TPOBDIIStatus
Dim bufferLength As Byte = 8
Dim buffer(bufferLength) As TPOBDIIResponse

' Send OBDII Service $04 request
result = OBDIIApi.ClearTroubleCodes(OBDIIApi.POBDII_USBBUS1, buffer, bufferLength)
If (result = TPOBDIIStatus.POBDII_ERROR_OK) Then
    MessageBox.Show("Request received responses.")
    ' search and remove unused responses
    For j As Integer = 0 To bufferLength - 1
        If (buffer(j).ERRORNR = TPOBDIIError.POBDII_R_NOT_USED) Then
            ' unused response, stop
            Exit For
        Else
            ' process response
    End If
Next
ElseIf (result = TPOBDIIStatus.POBDII_ERROR_NO_MESSAGE) Then
    MessageBox.Show("Request received no response.")
Else
    ' An error occurred
    MessageBox.Show("An error occured.")
End If
```
Pascal OO

```pascal
var
    result: TPOBDIITstatus;
    bufferLength: byte;
    buffer: array [0..7] of TPOBDIIResponse;
    j: integer;

begin
    bufferLength := 8;

    // Send OBDII Service $04 request
    result := TObdiiApi.ClearTroubleCodes(TObdiiApi.POBDDI_USBBUS1, @buffer, bufferLength);
    if (result = PODBII_ERROR_OK) then
        begin
            MessageBox(0, 'Request received responses.', 'Success', MB_OK);
            // search and remove unused responses
            for j := 0 To bufferLength - 1 do
                begin
                    if (buffer[j].ERRORNR = PODBII_R_NOT_USED) then
                        // unused response, stop
                        break
                    else
                        // process response
                        ;
                end
        end
    else if (result = PODBII_ERROR_NO_MESSAGE) then
        MessageBox(0, 'Request received no response.', 'Success', MB_OK)
    else
        // An error occurred
        MessageBox(0, 'An error occurred', 'Error', MB_OK)
end;
```

See also: TPOBDIIResponse on page 13

Plain function version: OBDII_ClearTroubleCodes

### 3.6.17 RequestTestResults

Sends an OBDII Service “Request On-Board Monitoring Results for Specific Monitored Systems” ($06) request into queue and waits to receive the responses. The purpose of this service is to allow access to the results for on-board diagnostic monitoring tests of specific components/systems that are continuously monitored (e.g. misfire monitoring for gasoline vehicles) and non-continuously monitored (e.g. catalyst system).

**Note:** This service includes functionality of service $05 “Request Oxygen Sensor Monitoring Test Results”. 
Syntax

Pascal OO

```pascal
class function RequestTestResults(
  CanChannel: TPOBDIICANHandle;
  OBDMid: TPOBDIIOBDMid;
  Data: PTPOBDIIMonitorData;
  DataLen: Byte
): TPOBDIIStatus;
```

C#

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestTestResults")]
public static extern TPOBDIIStatus RequestTestResults(
    [MarshalAs(UnmanagedType.U1)]
    TPOBDIICANHandle CanChannel,
    [MarshalAs(UnmanagedType.U1)]
    TPOBDIIOBDMid OBDMid,
    [In, Out] TPOBDIIMonitorData[] Data,
    byte DataLen);
```

C++ / CLR

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestTestResults")]
static TPOBDIIStatus RequestTestResults(
    [MarshalAs(UnmanagedType::U1)]
    TPOBDIICANHandle CanChannel,
    [MarshalAs(UnmanagedType::U1)]
    TPOBDIIOBDMid OBDMid,
    [Out] array<TPOBDIIMonitorData>^ Data,
    Byte DataLen);
```

Visual Basic

```visualbasic
<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_RequestTestResults")> _
Public Shared Function RequestTestResults( _
  ByVal CanChannel As TPOBDIICANHandle, _
  ByVal OBDMid As TPOBDIIOBDMid, _
  ByVal Data As TPOBDIIMonitorData(), _
  ByVal DataLen As Byte) As TPOBDIIStatus
End Function
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Obdmid</td>
<td>The On-Board Monitoring Identifier to request.</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIMonitoringData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_ERROR_NOT_INITIALIZED</td>
<td>Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application</td>
</tr>
</tbody>
</table>
POBDII_ERROR_NO_MESSAGE   | Indicates that no matching message was received
POBDII_ERROR_NO_MEMORY    | Failed to allocate a buffer to store the expected responses

**Remarks:** User should always check the error code of the generic response to assert that the response data is valid.

**Example**

The following example shows the use of the method `RequestTestResults` on the channel `POBDII_USBBUS1` with the `OBDMID 01`. Depending on the result, a message will be shown to the user.

**Note:** It is assumed that the channel was already initialized.

**C#**

```csharp
TPOBDIIStatus result;
byte bufferLength = 8;
TPOBDIIMonitorData[] buffer = new TPOBDIIMonitorData[8];

// Send OBDII Service $06 request with OBDMID 01
result = OBDIIApi.RequestTestResults(OBDIIApi.POBDII_USBBUS1, 0x01, buffer, bufferLength);
if (result == TPOBDIIStatus.POBDII_ERROR_OK)
{
    MessageBox.Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == TPOBDIIErr...
```visualbasic
Dim result As TPOBDIIStatus
Dim bufferLength As Byte = 8
Dim buffer(bufferLength) As TPOBDIIMonitorData
' Send OBDII Service $06 request with OBDMID 01
result = OBDIIApi.RequestTestResults(OBDIIApi.POBDII_USBBUS1, &H1, buffer, bufferLength)
If (result = TPOBDIIStatus.POBDII_ERROR_OK) Then
    MessageBox.Show("Request received responses.")
    ' search and remove unused responses
    For j As Integer = 0 To bufferLength - 1
        If (buffer(j).RESPONSE.ERRORNR = TPOBDIIError.POBDII_R_NOT_USED) Then
            ' unused response, stop
            Exit For
        Else
            ' process response
        End If
    Next
ElseIf (result = TPOBDIIStatus.POBDII_ERROR_NO_MESSAGE) Then
    MessageBox.Show("Request received no response.")
Else
    ' An error occurred
    MessageBox.Show("An error occurred.")
End If
```

```pascal
var
    result: TPOBDIIStatus;
    bufferLength: byte;
    buffer: array [0..7] of TPOBDIIMonitorData;
    j: integer;

begin
    bufferLength := 8;

    // Send OBDII Service $06 request with OBDMID 01
    result := TObdiiApi.RequestTestResults(TObdiiApi.POBDII_USBBUS1, $01, @buffer, bufferLength);
    if (result = POBDII_ERROR_OK) then
        begin
```

```
```pascal
MessageBox(0, 'Request received responses.', 'Success', MB_OK);
// search and remove unused responses
for j := 0 to bufferLength - 1 do
  begin
    if (buffer[j].RESPONSE.ERRORNR = POBDII_R_NOT_USED) then
      // unused response, stop
      break
    else
      // process response
    end
  end
else if (result = POBDII_ERROR_NO_MESSAGE) then
  MessageBox(0, 'Request received no response.', 'Success', MB_OK)
else
  // An error occurred
  MessageBox(0, 'An error occurred', 'Error', MB_OK)
end;

See also: TPOBDIIMonitorData on page 18

Plain function version: OBDII_RequestTestResults

3.6.18 RequestPendingTroubleCodes

Sends an OBDII Service “Request Emission-Related Diagnostic Trouble Codes Detected During Current or Last Completed Driving Cycle” ($07) request into queue and waits to receive the responses. The purpose of this service is to enable the external test equipment to obtain “pending” diagnostic trouble codes detected during current or last completed driving cycle for emission-related components/systems.

Syntax

Pascal

```pascal
class function RequestPendingTroubleCodes(
  CanChannel: TPOBDIICANHandle;
  Data: PTPOBDIIDTCData;
  DataLen: Byte
): TPOBDIIStatus;
```
C++ / CLR

```c++
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestPendingTroubleCodes")]
static TPOBDIIStatus RequestPendingTroubleCodes(
    [MarshalAs(UnmanagedType::U1)]
    TPOBDIICANHandle CanChannel,
    [Out] array<TPOBDIIDTCData>^ Data,
    Byte DataLen);
```

Visual Basic

```vbnet
<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_RequestPendingTroubleCodes")> _
Public Shared Function RequestPendingTroubleCodes( _
    ByVal CanChannel As TPOBDIICANHandle, _
    <[In](), Out()> ByVal Data As TPOBDIIDTCData(), _
    ByVal DataLen As Byte) As TPOBDIIStatus
End Function
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIDTC structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

- **POBDII_ERROR_NOT_INITIALIZED**: Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application
- **POBDII_ERROR_NO_MESSAGE**: Indicates that no matching message was received
- **POBDII_ERROR_NO_MEMORY**: Failed to allocate a buffer to store the expected responses

Remarks: User should always check the error code of the generic response to assert that the response data is valid.

Example

The following example shows the use of the method `RequestPendingTroubleCodes` on the channel `POBDII_USBBUS1`. Depending on the result, a message will be shown to the user.

ℹ️ Note: It is assumed that the channel was already initialized.

C#

```csharp
TPOBDIIStatus result;
byte bufferLength = 8;
TPOBDIIDTCData[] buffer = new TPOBDIIDTCData[8];

// Send OBDII Service $07 request
result = OBDIIApi.RequestPendingTroubleCodes(OBDIIApi.POBDII_USBBUS1, buffer, bufferLength);
if (result == TPOBDIIStatus.POBDII_ERROR_OK)
{
    MessageBox.Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
```
```csharp
if (buffer[j].RESPONSE.ERRORNR == TPOBDIIError.POBDII_R_NOT_USED)
{
    // unused response, stop
    break;
} else
{
    // process response
}
```

```csharp
else if (result == TPOBDIIStatus.POBDII_ERROR_NO_MESSAGE)
    MessageBox.Show("Request received no response.");
else
    // An error occurred
    MessageBox.Show("An error occurred.");
```

```
C++/CLR

TPOBDIIStatus result;
Byte bufferLength = 8;
array<TPOBDIIDTCData>^ buffer = gcnew array<TPOBDIIDTCData>(8);

// Send OBDII Service $07 request
result = OBDIIApi::RequestPendingTroubleCodes(OBDIIApi::POBDII_USBBUS1, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox::Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        } else
        {
            // process response
        }
    }
} else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox::Show("Request received no response.");
else
    // An error occurred
    MessageBox::Show("An error occurred.");
```

```
Visual Basic

Dim result As TPOBDIIStatus
Dim bufferLength As Byte = 8
Dim buffer(bufferLength) As TPOBDIIDTCData

' Send OBDII Service $07 request
result = OBDIIApi.RequestPendingTroubleCodes(OBDIIApi.POBDII_USBBUS1, buffer, bufferLength)
If (result = TPOBDIIStatus.POBDII_ERROR_OK) Then
    MessageBox.Show("Request received responses.")
```

' search and remove unused responses
For j As Integer = 0 To bufferLength - 1
    If (buffer(j).RESPONSE.ERRORNR = TPOBDIIError.POBDII_R_NOT_USED) Then
        ' unused response, stop
        Exit For
    Else
        ' process response
    End If
Next
ElseIf (result = TPOBDIISTatus.POBDII_ERROR_NO_MESSAGE) Then
    MessageBox.Show("Request received no response.")
Else
    ' An error occurred
    MessageBox.Show("An error occurred.")
End If

Pascal OO

var
    result: TPOBDIIStatus;
    bufferLength: byte;
    buffer: array [0..7] of TPOBDIITCData;
    j: integer;

begin
    bufferLength := 8;

    // Send OBDII Service $07 request
    result := TObdiiApi.RequestPendingTroubleCodes(TObdiiApi.POBDII_USBBUS1, @buffer, bufferLength);
    if (result = POBDII_ERROR_OK) then
        begin
            MessageBox(0, 'Request received responses.', 'Success', MB_OK);
            // search and remove unused responses
            for j := 0 To bufferLength - 1 do
                begin
                    if (buffer[j].RESPONSE.ERRORNR = POBDII_R_NOT_USED) then
                        // unused response, stop
                        break
                    else
                        // process response
                    end;
                end
            end
        else if (result = POBDII_ERROR_NO_MESSAGE) then
            MessageBox(0, 'Request received no response.', 'Success', MB_OK)
        else
            // An error occurred
            MessageBox(0, 'An error occurred', 'Error', MB_OK)
    end;

See also: [TPOBDIITCData] on page 17

Plain function version: OBDII_RequestPendingTroubleCodes
3.6.19 RequestControlOperation

Sends an OBDII Service “Request Control of On-Board System, Test or Component” ($08) request into queue and waits to receive the responses. The purpose of this service is to enable the external test equipment to control the operation of an on-board system, test or component.

Syntax

Pascal OO

```pascal
class function RequestControlOperation(  
  CanChannel: TPOBDIICANHandle;  
  Tid: TPOBDIITid;  
  Data: PTPOBDIIResponse;  
  DataLen: Byte  
): TPOBDIISTatus;
```

C#

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestControlOperation")]
public static extern TPOBDIISTatus RequestControlOperation(  
  [MarshalAs(UnmanagedType.U1)]  
  TPOBDIICANHandle CanChannel,  
  [MarshalAs(UnmanagedType.U1)]  
  TPOBDIITid Tid,  
  [In, Out]  
  TPOBDIIResponse[] Data,  
  byte DataLen);
```

C++ / CLR

```csharp
[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestControlOperation")]
static TPOBDIISTatus RequestControlOperation(  
  [MarshalAs(UnmanagedType::U1)]  
  TPOBDIICANHandle CanChannel,  
  [MarshalAs(UnmanagedType::U1)]  
  TPOBDIITid Tid,  
  [Out] array<TPOBDIIResponse>^ Data,  
  Byte DataLen);
```

Visual Basic

```vbnet
<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_RequestControlOperation")> _
Public Shared Function RequestControlOperation(  
  ByVal CanChannel As TPOBDIICANHandle, _  
  ByVal Tid As TPOBDIITid, _  
  ByVal Data As TPOBDIIResponse(), _  
  ByVal DataLen As Byte) As TPOBDIISTatus  
End Function
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Tid</td>
<td>The Test Identifier to request</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIIMonitoringData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>
Returns

The return value is a `TPOBDIISTatus` code. `POBDII_ERROR_OK` is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_ERROR_NOT_INITIALIZED</td>
<td>Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MESSAGE</td>
<td>Indicates that no matching message was received</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MEMORY</td>
<td>Failed to allocate a buffer to store the expected responses</td>
</tr>
</tbody>
</table>

Remarks: User should always check the error code of the generic response to assert that the response data is valid.

Example

The following example shows the use of the method `RequestControlOperation` on the channel `POBDII_USBBUS1` with `TID 01`. Depending on the result, a message will be shown to the user.

ℹ️ Note: It is assumed that the channel was already initialized.

C#  
```
TPOBDIISTatus result;
byte bufferLength = 8;
TPOBDIIResponse[] buffer = new TPOBDIIResponse[8];

// Send OBDII Service $08 request with TID 01
result = OBDIIApi.RequestControlOperation(OBDIIApi.POBDII_USBBUS1, 0x01, buffer, bufferLength);
if (result == TPOBDIISTatus.POBDII_ERROR_OK)
{
    MessageBox.Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].ERRORNR == TPOBDIIError.POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == TPOBDIISTatus.POBDII_ERROR_NO_MESSAGE)
    MessageBox.Show("Request received no response.");
else
    // An error occurred
    MessageBox.Show("An error occurred.");
```  

C++/CLR  
```
TPOBDIISTatus result;
Byte bufferLength = 8;
array<TPOBDIIResponse>^ buffer = gcnew array<TPOBDIIResponse>(8);

// Send OBDII Service $08 request with TID 01
result = OBDIIApi::RequestControlOperation(OBDIIApi::POBDII_USBBUS1, 0x01, buffer, bufferLength);
```
if (result == POBDII_ERROR_OK)
{
    MessageBox::Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox::Show("Request received no response.");
else
    // An error occurred
    MessageBox::Show("An error occurred.");

Visual Basic

Dim result As TPOBDIIStatus
Dim bufferLength As Byte = 8
Dim buffer(bufferLength) As TPOBDIIResponse

' Send OBDII Service $08 request with TID 01
result = OBDIIApi.RequestControlOperation(OBDIIApi.POBDII_USBBUS1, &H1, buffer, bufferLength)
If (result = TPOBDIIStatus.POBDII_ERROR_OK) Then
    MessageBox.Show("Request received responses.")
    ' search and remove unused responses
    For j As Integer = 0 To bufferLength - 1
        If (buffer(j).ERRORNR = TPOBDIIError.POBDII_R_NOT_USED) Then
            ' unused response, stop
            Exit For
        Else
            ' process response
    End If
Next
ElseIf (result = TPOBDIIStatus.POBDII_ERROR_NO_MESSAGE) Then
    MessageBox.Show("Request received no response.")
Else
    ' An error occurred
    MessageBox.Show("An error occurred.")
End If
Pascal OO

```pascal
var
result: TPOBDIIStatus;
bufferLength: byte;
buffer: array [0..7] of TPOBDIIResponse;
j: integer;

begin

bufferLength := 8;

// Send OBDII Service $08 request with TID 01
result := TObdiiApi.RequestControlOperation(TObdiiApi.POBDII_USBBUS1, $01, @buffer, bufferLength);
if (result = POBDII_ERROR_OK) then

begin

MessageBox(0, 'Request received responses.', 'Success', MB_OK);
// search and remove unused responses
for j := 0 To bufferLength - 1 do

begin
if (buffer[j].ERRORNR = POBDII_R_NOT_USED) then

// unused response, stop
break
else

// process response
;
end
end
else if (result = POBDII_ERROR_NO_MESSAGE) then

MessageBox(0, 'Request received no response.', 'Success', MB_OK)
else

// An error occurred
MessageBox(0, 'An error occurred', 'Error', MB_OK)
end;
```

See also: TOBDIIResponse on page 13

Plain function version: OBDII_RequestControlOperation

### 3.6.20 RequestVehicleInformation

Sends an OBDII Service “RequestVehicleInformation” ($09) request into queue and waits to receive the responses. The purpose of this service is to enable the external test equipment to request vehicle-specific vehicle information such as Vehicle Identification Number (VIN) and Calibration IDs.

**Syntax**

**Pascal OO**

```pascal
class function RequestVehicleInformation(
    CanChannel: TPOBDIIICANHandle;
    InfoType: TPOBDIIInfoType;
    Data: PTPCDBDIIInfoData;
    DataLen: Byte): TPOBDIIStatus;
```
### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>InfoType</td>
<td>The InfoType Identifier to request</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIMonitoringData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

### Returns

The return value is a `TPOBDIIStatus` code. `POBDII_ERROR_OK` is returned on success. The typical errors in case of failure are:

- **POBDII_ERROR_NOT_INITIALIZED**: Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application.
- **POBDII_ERROR_NO_MESSAGE**: Indicates that no matching message was received.
- **POBDII_ERROR_NO_MEMORY**: Failed to allocate a buffer to store the expected responses.

**Remarks:** User should always check the error code of the generic response to assert that the response data is valid.
### Example

The following example shows the use of the method [RequestVehicleInformation](#) on the channel POBDII_USBBUS1 with InfoType 02 (i.e. request Vehicle Identification Number). Depending on the result, a message will be shown to the user.

**Note:** It is assumed that the channel was already initialized.

**C#**

```csharp
TPOBDIIStatus result;
byte bufferLength = 8;
TPOBDIIInfoData[] buffer = new TPOBDIIInfoData[8];

// Send OBDII Service $09 request with InfoType 02
result = OBDIIApi.RequestVehicleInformation(OBDIIApi.POBDII_USBBUS1, 0x02, buffer, bufferLength);
if (result == TPOBDIIStatus.POBDII_ERROR_OK)
{
    MessageBox.Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == TPOBDIIError.POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {  // process response
        }
    }
}
else if (result == TPOBDIIStatus.POBDII_ERROR_NO_MESSAGE)
    MessageBox.Show("Request received no response.");
else
    // An error occurred
    MessageBox.Show("An error occured.");
```

**C++/CLR**

```csharp
TPOBDIIStatus result;
Byte bufferLength = 8;
array<TPOBDIIInfoData>^ buffer = gcnew array<TPOBDIIInfoData>(8);

// Send OBDII Service $09 request with InfoType 02
result = OBDIIApi::RequestVehicleInformation(OBDIIApi::POBDII_USBBUS1, 0x02, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox::Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {  // process response
        }
    }
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox::Show("Request received no response.");
else
    // An error occurred
    MessageBox::Show("An error occurred.");
```
{  // process response
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox::Show("Request received no response.");
else
    // An error occurred
    MessageBox::Show("An error occurred.");

Visual Basic

Dim result As TPOBDIIStatus
Dim bufferLength As Byte = 8
Dim buffer(bufferLength) As TPOBDIIInfoData
'
'Send OBDII Service $09 request with InfoType 02
result = OBDIIApi.RequestVehicleInformation(OBDIIApi.POBDII_USBBUS1, &H2, buffer, bufferLength)
If (result = TPOBDIIStatus.POBDII_ERROR_OK) Then
    MessageBox.Show("Request received responses.")
    ' search and remove unused responses
    For j As Integer = 0 To bufferLength - 1
        If (buffer(j).RESPONSE.ERRORNR = TPOBDIIError.POBDII_R_NOT_USED) Then
            ' unused response, stop
            Exit For
        Else
            ' process response
        End If
    Next
ElseIf (result = TPOBDIIStatus.POBDII_ERROR_NO_MESSAGE) Then
    MessageBox.Show("Request received no response.")
Else
    ' An error occurred
    MessageBox.Show("An error occurred.")
End If

Pascal OO

var
    result: TPOBDIIStatus;
    bufferLength: byte;
    buffer: array [0..7] of TPOBDIIInfoData;
    j: integer;
begin
    bufferLength := 8;

    // Send OBDII Service $09 request with InfoType 02
    result := TObdiiApi.RequestVehicleInformation(TObdiiApi.POBDII_USBBUS1, $02, @buffer, bufferLength);
    if (result = POBDII_ERROR_OK) then
        begin
            MessageBox(0, 'Request received responses.', 'Success', MB_OK);
            // search and remove unused responses
            for j := 0 To bufferLength - 1 do
                begin
                    // etc.
if (buffer[j].RESPONSE.ERRORNR = POBDII_R_NOT_USED) then
    // unused response, stop
    break
else
    // process response
    ;
end
else if (result = POBDII_ERROR_NO_MESSAGE) then
    MessageBox(0, 'Request received no response.', 'Success', MB_OK)
else
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK)
end;

See also: TPOBDIIInfoData on page 20

Plain function version: OBDII_RequestVehicleInformation

3.6.21 RequestPermanentTroubleCodes

Sends an OBDII Service “Request Emission-Related Diagnostic Trouble Codes with Permanent Status” ($0A) request into queue and waits to receive the responses. The purpose of this service is to enable the external test equipment to obtain all DTCs with “permanent DTC” status.

Syntax

Pascal OO

class function RequestPermanentTroubleCodes(
    CanChannel: TPOBDIICANHandle;
    Data: PTPOBDIIDTCData;
    DataLen: Byte): TPOBDIIStatus;

C#

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestPermanentTroubleCodes")]
public static extern TPOBDIIStatus RequestPermanentTroubleCodes(
    [MarshalAs(UnmanagedType.U1)]
    TPOBDIICANHandle CanChannel,
    [MarshalAs(UnmanagedType.U1)]
    TPOBDIIDTCData[] Data,
    byte DataLen);

C++ / CLR

[DllImport("PCAN-OBDII.dll", EntryPoint = "OBDII_RequestPermanentTroubleCodes")]
static TPOBDIIStatus RequestPermanentTroubleCodes(
    [MarshalAs(UnmanagedType::U1)]
    TPOBDIICANHandle CanChannel,
    [MarshalAs(UnmanagedType::U1)]
    array<TPOBDIIDTCData>^ Data,
    Byte DataLen);
Visual Basic

<DllImport("PCAN-OBDII.dll", EntryPoint:="OBDII_RequestPermanentTroubleCodes")> 
Public Shared Function RequestPermanentTroubleCodes( 
<MarshalAs(UnmanagedType.U1)> 
ByVal CanChannel As TPOBDIICANHandle, 
<[In](), Out()> ByVal Data As TPOBDIIDTCData(), 
ByVal DataLen As Byte) As TPOBDIIStatus
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIDTCData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

- POBDII_ERROR_NOT_INITIALIZED Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application
- POBDII_ERROR_NO_MESSAGE Indicates that no matching message was received
- POBDII_ERROR_NO_MEMORY Failed to allocate a buffer to store the expected responses

Remarks: User should always check the error code of the generic response to assert that the response data is valid.

Example

The following example shows the use of the method RequestPermanentTroubleCodes on the channel POBDII_USBBUS1. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```csharp
TPOBDIIStatus result;
byte bufferLength = 8;
TPOBDIIDTDData[] buffer = new TPOBDIIDTDData[8];

// Send OBDII Service $0A
result = OBDIIApi.RequestPermanentTroubleCodes(OBDIIApi.POBDII_USBBUS1, buffer, bufferLength);
if (result == TPOBDIIStatus.POBDII_ERROR_OK)
{
    MessageBox.Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == TPOBDIIError.POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
```
C++/CLR

TPOBDIIStatus result;
Byte bufferLength = 8;
array<TPOBDIIDTCData>^ buffer = gcnew array<TPOBDIIDTCData>(8);

// Send OBDII Service $0A
result = OBDIIApi::RequestPermanentTroubleCodes(OBDIIApi::POBDII_USBBUS1, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox::Show("Request received responses.");
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox::Show("Request received no response.");
else
    // An error occurred
    MessageBox::Show("An error occurred.");

Visual Basic

Dim result As TPOBDIIStatus
Dim bufferLength As Byte = 8
Dim buffer(bufferLength) As TPOBDIIDTCData

' Send OBDII Service $0A
result = OBDIIApi.RequestPermanentTroubleCodes(OBDIIApi.POBDII_USBBUS1, buffer, bufferLength)
If (result = TPOBDIIStatus.POBDII_ERROR_OK) Then
    MessageBox.Show("Request received responses.")
    ' search and remove unused responses
    For j As Integer = 0 To bufferLength - 1
        If (buffer(j).RESPONSE.ERRORNR = TPOBDIIError.POBDII_R_NOT_USED) Then
            ' unused response, stop
            Exit For
        Else
            ' process response
        End If
    Next
ElseIf (result = TPOBDIIStatus.POBDII_ERROR_NO_MESSAGE) Then
MessageBox.Show("Request received no response.")
Else
  ' An error occurred
  MessageBox.Show("An error occurred.")
End If

### Pascal OO

```pascal
var
  result: TPOBDIIStatus;
  bufferLength: byte;
  buffer: array [0..7] of TPOBDIIDTCData;
  j: integer;

begin
  bufferLength := 8;

  // Send OBDII Service $0A
  result := TObdiiApi.RequestPermanentTroubleCodes(TObdiiApi.POBDII_USBBUS1, @buffer, bufferLength);
  if (result = POBDII_ERROR_OK) then
    begin
      MessageBox(0,'Request received responses.', 'Success', MB_OK);
      // search and remove unused responses
      for j := 0 To bufferLength - 1 do
        begin
          if (buffer[j].RESPONSE.ERRORNR = POBDII_R_NOT_USED) then
            // unused response, stop
            break
          else
            // process response
            ;
        end
    end
  else if (result = POBDII_ERROR_NO_MESSAGE) then
    MessageBox(0,'Request received no response.', 'Success', MB_OK)
  else
    // An error occurred
    MessageBox(0,'An error occurred', 'Error', MB_OK)
end;
```

See also: [TPOBDIIDTCData] on page 17

Plain function version: [OBDII_RequestPermanentTroubleCodes]
### 3.7 Functions

The functions of the PCAN OBDII API are divided into 4 groups of functionality.

#### Connection

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBDII_Initialize</td>
<td>Initializes a POBDII Channel</td>
</tr>
<tr>
<td>OBDII_Uninitialize</td>
<td>Uninitializes a POBDII Channel</td>
</tr>
</tbody>
</table>

#### Configuration

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBDII_SetValue</td>
<td>Sets a configuration or information value within a POBDII Channel</td>
</tr>
</tbody>
</table>

#### Information

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBDII_GetValue</td>
<td>Retrieves information from a POBDII Channel</td>
</tr>
<tr>
<td>OBDII_GetStatus</td>
<td>Retrieves the current BUS status of a POBDII Channel</td>
</tr>
<tr>
<td>OBDII_GetUnitAndScaling</td>
<td>Retrieves the unit and scaling information for a specified Unit and Scaling ID</td>
</tr>
</tbody>
</table>

#### Communication

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBDII_Reset</td>
<td>Resets the receive and transmit queues of a POBDII Channel</td>
</tr>
<tr>
<td>OBDII_RequestCurrentData</td>
<td>Sends an OBDII Service $01 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>OBDII_RequestFreezeFrameData</td>
<td>Sends an OBDII Service $02 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>OBDII_RequestStoredTroubleCodes</td>
<td>Sends an OBDII Service $03 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>OBDII_ClearTroubleCodes</td>
<td>Sends an OBDII Service $04 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>OBDII_RequestTestResults</td>
<td>Sends an OBDII Service $06 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>OBDII_RequestPendingTroubleCodes</td>
<td>Sends an OBDII Service $07 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>OBDII_RequestControlOperation</td>
<td>Sends an OBDII Service $08 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>OBDII_RequestVehicleInformation</td>
<td>Sends an OBDII Service $09 request into queue and waits to receive the responses</td>
</tr>
<tr>
<td>OBDII_RequestPermanentTroubleCodes</td>
<td>Sends an OBDII Service $0A request into queue and waits to receive the responses</td>
</tr>
</tbody>
</table>
3.7.1 OBDII.Initialize

Initializes a POBDII Channel.

Syntax

C++

```cpp
TPOBDIIStatus __stdcall OBDII_Initialize(
    TPOBDIICANHandle CanChannel,
    TPOBDII_BAUDRATE_AUTODETECT,  
    TPOBDII_BAUDRATE_AUTODETECT,  
    DWORD IOPort _DEF_ARG,         
    WORD Interrupt _DEF_ARG);      
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Baudrate</td>
<td>The speed for the communication (see TPOBDII_BAUDRATE_AUTODETECT)</td>
</tr>
<tr>
<td>HwType</td>
<td>The type of hardware (see TPOBDII_BAUDRATE_AUTODETECT)</td>
</tr>
<tr>
<td>IOPort</td>
<td>The I/O address for the parallel port</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Interrupt number of the parallel port</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

- POBDII_ERROR_ALREADY_INITIALIZED: Indicates that the desired POBDII Channel is already in use.
- POBDII_ERROR_UNSUPPORTED_ECUS: Indicates that no ECUs where found during the autodetection mechanism.
- POBDII_ERROR_OVERFLOW: Maximum number of initialized channels reached.
- POBDII_ERROR_CAN_ERROR: This error flag states that the error is composed of a more precise PCAN-Basic error.

Remarks: As indicated by its name, the OBDII.Initialize function initiates a POBDII Channel, preparing it for communication within the CAN bus connected to it. Calls to the other functions will fail if they are used with a Channel handle, different than POBDII_NONEBUS, that has not been initialized yet. Each initialized channel should be released when it is not needed anymore.

Initializing a POBDII Channel means:

- to reserve the Channel for the calling application/process
- to detect the baudrate of the CAN bus (if requested with TPOBDII_BAUDRATE_AUTODETECT value)
- to allocate channel resources, like receive and transmit queues
- to forward initialization to PCAN-UDS, PCAN-ISO-TP API, and PCAN-Basic API, hence registering/connecting the Hardware denoted by the channel handle

The Initialization process will fail if an application tries to initialize a POBDII-Channel that has already been initialized within the same process.

Take in consideration that initializing a channel causes a reset of the CAN hardware. In this way errors like BUSOFF, BUSHEAVY, and BUSLIGHT, are removed.

The PCAN-OBD-2 API use the same function for initializations of both, Plug-And-Play and Not-Plug-And-Play hardware. The OBDII.Initialize function has three additional parameters that are only for the connection
of Non-Plug-And-Play hardware. With Plug-And-Play hardware, however, only two parameters are to be supplied. The remaining three are not evaluated.

**Example**

The following example shows the initialize and uninitialize processes for a Non-Plug-And-Play channel (channel 1 of the PCAN-DNG).

ℹ️ **Note:** The initialization specifies a baudrate thus skipping the auto-detection mechanism.

```cpp
TPOBDIISstatus result;

// The Plug & Play Channel (PCAN-PCI) is initialized
result = OBDII_Initialize(POBDII_PCIBUS2);
if (result != POBDII_ERROR_OK)
    MessageBox(NULL, "Initialization failed", "Error", MB_OK);
else
    MessageBox(NULL, "PCAN-PCI (Ch-2) was initialized", "Success", MB_OK);

// All initialized channels are released
OBDII_Uninitialize(POBDII_NONEBUS);
```

See also: [OBDII_Uninitialize](#) below, [OBDII_GetValue](#) on page 106, Understanding PCAN-OBD-2 on page 6

**Class-method:** Initialize

### 3.7.2 OBDII_Uninitialize

Uninitializes a POBDII Channel.

**Syntax**

**C++**

```cpp
TPOBDIISstatus __stdcall OBDII_Uninitialize(
    TPOBDIICANHandle CanChannel);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
</tbody>
</table>

**Returns**

The return value is a TPOBDIISstatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

- POBDII_ERROR_NOT_INITIALIZED Indicates that the given POBDII channel cannot be uninitialized because it was not found in the list of reserved channels of the calling application

**Remarks:** A POBDII Channel can be released using one of these possibilities:

- **Single-Release:** Given a handle of a POBDII channel initialized before with the function OBDII_Initialize. If the given channel can not be found then an error is returned
- **Multiple-Release:** Giving the handle value POBDII_NONEBUS which instructs the API to search for all channels initialized by the calling application and release them all. This option cause no errors if no hardware were uninitialized
Example

The following example shows the initialize and uninitialize processes for a Plug-And-Play channel (channel 2 of a PCAN-PCI hardware).

C++:

```c++
TPOBDIIStatus result;

// The Non-Plug & Play Channel (PCAN-DNG) is initialized
result = OBDII_Initialize(POBDII_PCIBUS2, POBDII_BAUDRATE_500K, 0, 0, 0);
if (result != POBDII_ERROR_OK)
    MessageBox(NULL, "Initialization failed", "Error", MB_OK);
else
    MessageBox(NULL, "PCAN-PCI (Ch-2) was initialized", "Success", MB_OK);

// Release channel
OBDII_Uninitialize(POBDII_PCIBUS2);
```

See also: `OBDII_Initialize` on page 103

Class-method: `Uninitialize`

### 3.7.3 OBDII_SetValue

Sets a configuration or information value within a POBDII Channel.

**Syntax**

C++

```c++
TPOBDIIStatus __stdcall OBDII_SetValue(
    TPOBDIICANHandle CanChannel,
    TPOBDIIParameter Parameter,
    void* Buffer,
    DWORD BufferLength);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the value to be set (see TPOBDIIParameter on page 31)</td>
</tr>
<tr>
<td>Buffer</td>
<td>The buffer containing the value to be set</td>
</tr>
<tr>
<td>BufferLength</td>
<td>The length in bytes of the given buffer</td>
</tr>
</tbody>
</table>

**Returns**

The return value is a `TPOBDIIStatus` code. `POBDII_ERROR_OK` is returned on success. The typical errors in case of failure are:

| `POBDII_ERROR_NOT_INITIALIZED` | Indicates that the given POBDII channel cannot be uninitialized because it was not found in the list of reserved channels of the calling application |
| `POBDII_ERROR_WRONG_PARAM`    | Indicates that the parameters passed to the method are invalid. Check the value of 'Parameter' and assert it is compatible with an integer buffer |

**Remarks:** Use the method `SetValue` to set configuration information or environment values of a POBDII Channel. Note that any calls with non OBDII parameters (i.e. `TPOBDIIParameter`) will be forwarded to PCAN-UDS, PCAN-ISO-TP API, and PCAN-Basic API.
More information about the parameters and values that can be set can be found in Parameter Value Definitions.

**Example**
The following example shows the use of the function `OBDII_SetValue` on the channel `POBDII_PCIBUS2` to enable debug mode.

**Note:** It is assumed that the channel was already initialized.

```cpp
TPOBDIIStatus result;
unsigned int iBuffer = 0;

// Enable CAN DEBUG mode
iBuffer = POBDII_DEBUG_CAN;
result = OBDII_SetValue(POBDII_PCIBUS2, POBDII_PARAM_DEBUG, &iBuffer,
sizeof(unsigned int));
if (result != POBDII_ERROR_OK)
    MessageBox(NULL, "Failed to set value", "Error", MB_OK);
else
    MessageBox(NULL, "Value changed successfully ", "Success", MB_OK);
```

See also: [OBDII_GetValue](#) below

**Class-method:** [GetValue](#)

### 3.7.4 OBDII_GetValue
Retrieves information from a POBDII Channel.

**Syntax**

```cpp
TPOBDIIStatus __stdcall OBDII_GetValue(
    TPOBDIICANHandle CanChannel,
    TPOBDIIParameter Parameter,
    void* Buffer,
    DWORD BufferLength);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the value to be set (see TPOBDIIParameter on page 31)</td>
</tr>
<tr>
<td>Buffer</td>
<td>The buffer to return the required value</td>
</tr>
<tr>
<td>BufferLength</td>
<td>The length in bytes of the given buffer</td>
</tr>
</tbody>
</table>

**Returns**
The return value is a `TPOBDIIStatus` code. `POBDII_ERROR_OK` is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_ERROR_NOT_INITIALIZED</td>
<td>Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application</td>
</tr>
<tr>
<td>POBDII_ERROR_WRONG_PARAM</td>
<td>Indicates that the parameters passed to the method are invalid. Check the value of ‘Parameter’ and assert it is compatible with a string buffer</td>
</tr>
</tbody>
</table>
Example

The following example shows the use of the function `OBDII_GetValue` on the channel `POBDII_USBBUS1` to retrieve the number of connected and responding ECUs on the CAN bus. Depending on the result, a message will be shown to the user.

**Note:** It is assumed that the channel was already initialized.

### C++

```cpp
TPOBDIICANHandle CanChannel = POBDII_USBBUS1;
TPOBDIIStatus result;
unsigned int iBuffer = 0;
char strMsg[256];

// Get the number of connected ECUs
result = OBDII_GetValue(CanChannel, POBDII_PARAM_AVAILABLE_ECUS, &iBuffer, sizeof(unsigned int));
if (result != POBDII_ERROR_OK)
    MessageBox(NULL, "Failed to get value", "Error", MB_OK);
else
{
    sprintf(strMsg, "%d", iBuffer);
    MessageBox(NULL, strMsg, "Success", MB_OK);
}
```

See also: [OBDII_SetValue](#) on page 105, [TPOBDIIParameter](#) on page 31, [Parameter Value Definitions](#) on page 124

### Class-method: `GetValue`

#### 3.7.5 OBDII_GetStatus

Gets the current BUS status of a POBDII Channel.

### Syntax

**C++**

```cpp
TPOBDIIStatus __stdcall OBDII_GetStatus(
    TPOBDIICANHandle CanChannel);
```

### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
</tbody>
</table>

### Returns

The return value is a `TPOBDIIStatus` code. `POBDII_ERROR_OK` is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_ERROR_OK</td>
<td>Indicates that the status of the given POBDII channel is OK</td>
</tr>
<tr>
<td>POBDII_ERROR_BUSLIGHT</td>
<td>Indicates a bus error within the given POBDII Channel. The hardware is in bus-light status</td>
</tr>
<tr>
<td>POBDII_ERROR_BUSHEAVY</td>
<td>Indicates a bus error within the given POBDII Channel. The hardware is in bus-heavy status</td>
</tr>
<tr>
<td>POBDII_ERROR_BUSOFF</td>
<td>Indicates a bus error within the given POBDII Channel. The hardware is in bus-off status</td>
</tr>
<tr>
<td>POBDII_ERROR_NOT_INITIATED:</td>
<td>Indicates that the given POBDII channel cannot be used because it was not found in the list of reserved channels of the calling application</td>
</tr>
</tbody>
</table>
Remarks: When the hardware status is bus-off, an application cannot communicate anymore. Consider using the PCAN-Basic property [PCAN_BUSOFF_AUTORESET] which instructs the API to automatically reset the CAN controller when a bus-off state is detected.

Another way to reset errors like bus-off, bus-heavy and bus-light, is to uninitialize and initialize again the channel used. This causes a hardware reset.

Example

The following example shows the use of the function [OBDII_GetStatus] on the channel [POBDII_PCIBUS1]. Depending on the result, a message will be shown to the user.

-note: It is assumed that the channel was already initialized.

C++

```c++
TPOBDIIStatus result;

// Check the status of the PCI Channel
result = OBDII_GetStatus(POBDII_PCIBUS1);
switch (result) {
    case POBDII_ERROR_BUSLIGHT:
        MessageBox(NULL, "PCAN-PCI (Ch-1): Handling a BUS-LIGHT status...", "Success", MB_OK);
        break;
    case POBDII_ERROR_BUSHEAVY:
        MessageBox(NULL, "PCAN-PCI (Ch-1): Handling a BUS-HEAVY status...", "Success", MB_OK);
        break;
    case POBDII_ERROR_BUSOFF:
        MessageBox(NULL, "PCAN-PCI (Ch-1): Handling a BUS-OFF status...", "Success", MB_OK);
        break;
    case POBDII_ERROR_OK:
        MessageBox(NULL, "PCAN-PCI (Ch-1): Status is OK", "Success", MB_OK);
        break;
    default:
        // An error occurred:
        MessageBox(NULL, "Failed to retrieve status", "Error", MB_OK);
        break;
}
```

Class-method: [getStatus]

### 3.7.6 OBDII_Reset

Resets the receive and transmit queues of a POBDII Channel.

Syntax

C++

```c++
TPOBDIIStatus ___stdcall OBDII_Reset(
    TPOBDIICANHandle CanChannel);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
</tbody>
</table>
Returns
The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

- POBDII_ERROR_NOT_INITIALIZED Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application.

Remarks: Calling this function ONLY clears the queues of a Channel. A reset of the CAN controller doesn’t take place.

Example
The following example shows the use of the function OBDII_Reset on the channel POBDII_PCIBUS1. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

```cpp
TPOBDIIStatus result;

// The PCI Channel is reset
result = OBDII_Reset(POBDII_PCIBUS1);
if (result != POBDII_ERROR_OK)
{
    // An error occurred
    MessageBox(NULL, "An error occurred", "Error", MB_OK);
}
else
    MessageBox(NULL, "PCAN-PCI (Ch-1) was reset", "Success", MB_OK);
```

See also: OBDII_Uninitialize on page 104

Class-method: Reset

3.7.7 OBDII_GetUnitAndScaling
Retrieves information from a Unit and Scaling Identifier.

Syntax

```cpp
TPOBDIIStatus __stdcall OBDII_GetUnitAndScaling(
    BYTE id,
    TPOBDIIUnitAndScaling * unitAndScaling);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>The Unit and Scaling Identifier to retrieve information from</td>
</tr>
<tr>
<td>UnitAndScaling</td>
<td>A buffer to store the TPOBDIIUnitAndScaling information</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

- POBDII_ERROR_NOT_INITIALIZED Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application.
Example
The following example shows the use of the function OBDII_GetUnitAndScaling with the Voltage ID (0x0A). Depending on the result, a message will be shown to the user.

⚠️ Note: It is assumed that the channel was already initialized.

C++

```cpp
TPOBDIIStatus result;
TPOBDIIUnitAndScaling unitAndScaling;
char strMsg[256];

// Get Unit and Scaling information
result = OBDII_GetUnitAndScaling(0x0A, &unitAndScaling);
if (result != POBDII_ERROR_OK)
{
    // An error occurred
    MessageBox(NULL, "An error occurred", "Error", MB_OK);
}
else
{
    sprintf(strMsg, "Unit: %s", unitAndScaling.UNIT);
    MessageBox(NULL, strMsg, "Success", MB_OK);
}
```

See also: OBDII_RequestTestResults on page 115

Class-method: OBDII_GetUnitAndScaling

3.7.8 OBDII_RequestCurrentData

Sends an OBDII Service “Request Current Powertrain Diagnostic Data” ($01) request into queue and waits to receive the responses. The purpose of this service is to allow access to current emission-related data values, including analogue inputs and outputs, digital inputs and outputs, and system status information.

Syntax

C++

```cpp
TPOBDIIStatus __stdcall OBDII_RequestCurrentData(
    TPOBDIICANHandle CanChannel,
    TPOBDIIPid Pid,
    TPOBDIIParamData* Data,
    BYTE DataLen);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Pid</td>
<td>The Parameter Identifier to request</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIParamData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:
### POBDII_ERROR_NOT_INITIALIZED
Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application

### POBDII_ERROR_NO_MESSAGE
Indicates that no matching message was received

### POBDII_ERROR_NO_MEMORY
Failed to allocate a buffer to store the expected responses

**Remarks:** User should always check the error code of the generic response to assert that the response data is valid.

**Example**

The following example shows the use of the method function `OBDII_RequestCurrentData` on the channel `POBDII_USBBUS1` with PID 01. Depending on the result, a message will be shown to the user. If responses are received, a loop is set to handle valid responses.

**Note:** It is assumed that the channel was already initialized.

**C++**

```cpp
TPOBDIIStatus result;
BYTE bufferLength = 8;
TPOBDIIParamData buffer[8];

// Send OBDII Service $01 request
result = OBDII_RequestCurrentData(POBDII_USBBUS1, 0x01, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox(NULL, "Request received responses.", "Success", MB_OK);
    // loop through responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox(NULL, "Request received no response.", "Success", MB_OK);
else
    // An error occurred
    MessageBox(NULL, "An error occurred", "Error", MB_OK);
```

**See also:** [TPOBDIIParamData](#) on page 14

**Class-method:** `RequestCurrentData`

### 3.7.9 OBDII_RequestFreezeFrameData

Sends an OBDII Service “Request Powertrain Freeze Frame Data” ($02) request into queue and waits to receive the responses. The purpose of this service is to allow access to emission-related data values in a freeze frame.

**Syntax**
C++

TPOBDIStatus __stdcall OBDII_RequestFreezeFrameData(
    TPOBDIICANHandle CanChannel,
    TPOBDIIPid Pid,
    BYTE Frame,
    TPOBDIIParamData* Data,
    BYTE DataLen);

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Pid</td>
<td>The Parameter Identifier to request</td>
</tr>
<tr>
<td>Frame</td>
<td>The Freeze Frame number</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIParamData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

POBDII_ERROR_NOT_INITIALIZED  | Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application
POBDII_ERROR_NO_MESSAGE     | Indicates that no matching message was received
POBDII_ERROR_NO_MEMORY      | Failed to allocate a buffer to store the expected responses

Remarks: User should always check the error code of the generic response to assert that the response data is valid.

Example

The following example shows the use of the function OBDII_RequestFreezeFrameData on the channel POBDII_USBBUS1 with PID 02 and Freeze Frame 00 (i.e. read the DTC that caused the freeze frame data to be stored). Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.


```c

// process response

}

else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox(NULL, "Request received no response.", "Success", MB_OK);
else
    // An error occurred
    MessageBox(NULL, "An error occurred", "Error", MB_OK);

See also: [TPOBDIIParamData on page 14]

Class-method: RequestFreezeFrameData

3.7.10  OBDII_RequestStoredTroubleCodes

Sends an OBDII Service “Request Emission-Related Diagnostic Information” ($03) request into queue and
waits to receive the responses. The purpose of this service is to enable the external test equipment to
obtain “confirmed” emission-related DTCs.

Syntax

C++

TPOBDIIStatus __stdcall OBDII_RequestStoredTroubleCodes(
    TPOBDIICANHandle CanChannel,
    TPOBDIIDTCData* Data,
    BYTE DataLen);

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIDTCData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a [TPOBDIIStatus] code. POBDII_ERROR_OK is returned on success. The typical errors in case of
failure are:

- POBDII_ERROR_NOT_INITIALIZED: Indicates that the given POBDII channel was not found in the list of initialized
  channels of the calling application
- POBDII_ERROR_NO_MESSAGE: Indicates that no matching message was received
- POBDII_ERROR_NO_MEMORY: Failed to allocate a buffer to store the expected responses

Remarks: User should always check the error code of the generic response to assert that the response
data is valid.

Example

The following example shows the use of the function OBDII_RequestStoredTroubleCodes on the channel
POBDII_USBBUS1. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.
C++

```c++
TPOBDIStatus result;
BYTE bufferLength = 8;
TPOBDIIDTCData buffer[8];

// Send OBDII Service $03 request
result = OBDII_RequestStoredTroubleCodes(POBDII_USBBUS1, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox(NULL, "Request received responses.", "Success", MB_OK);
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox(NULL, "Request received no response.", "Success", MB_OK);
else
    // An error occurred
    MessageBox(NULL, "An error occurred", "Error", MB_OK);
```

See also: [TPOBDIIDTCData](#) on page 17

Class-method: [RequestStoredTroubleCodes](#)

### 3.7.11 OBDII_ClearTroubleCodes

Sends an OBDII Service “Clear/Reset Emission-Related Diagnostic Information” ($04) request into queue and waits to receive the responses. The purpose of this service is to provide a means for the external test equipment to command ECUs to clear all emission-related diagnostic information.

**Syntax**

```c++
TPOBDIStatus __stdcall OBDII_ClearTroubleCodes(
    TPOBDIICANHandle CanChannel,
    TPOBDIResponse* Response,
    BYTE DataLen);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIResponse structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>
Returns
The return value is a `TPOBDIIStatus` code. `POBDII_ERROR_OK` is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_ERROR_NOT_INITIALIZED</td>
<td>Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MESSAGE</td>
<td>Indicates that no matching message was received</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MEMORY</td>
<td>Failed to allocate a buffer to store the expected responses</td>
</tr>
</tbody>
</table>

Remarks: User should always check the error code of the generic response to assert that the response data is valid.

Example
The following example shows the use of the function `OBDII_ClearTroubleCodes` on the channel `POBDII_USBBUS1`. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C++

```c++
TPOBDIIStatus result;
BYTE bufferLength = 8;
TPOBDIIResponse buffer[8];

// Send OBDII Service $04 request
result = OBDII_ClearTroubleCodes(POBDII_USBBUS1, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox(NULL, "Request received responses.", "Success", MB_OK);
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox(NULL, "Request received no response.", "Success", MB_OK);
else
    // An error occurred
    MessageBox(NULL, "An error occurred", "Error", MB_OK);
```

See also: `TOBDIIResponse` on page 13

Class-method: `ClearTroubleCodes`

### 3.7.12 OBDII_RequestTestResults
Sends an OBDII Service “Request On-Board Monitoring Results for Specific Monitored Systems” ($06) request into queue and waits to receive the responses. The purpose of this service is to allow access to the results for on-board diagnostic monitoring tests of specific components/systems that are
continuously monitored (e.g. misfire monitoring for gasoline vehicles) and non-continuously monitored (e.g. catalyst system).

**Note:** This service includes functionality of service $05 “Request Oxygen Sensor Monitoring Test Results”.

**Syntax**

**C++**

```cpp
TPOBDIStatus __stdcall OBDII_RequestTestResults(
    TPOBDIICANHandle CanChannel,
    TPOBDIIOBDMid OBDMid,
    TPOBDIIMonitorData* Data,
    BYTE DataLen);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Obdmid</td>
<td>The On-Board Monitoring Identifier to request</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIMonitoringData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

**Returns**

The return value is a [TPOBDIStatus] code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

- **POBDII_ERROR_NOT_INITIALIZED**: Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application.
- **POBDII_ERROR_NO_MESSAGE**: Indicates that no matching message was received.
- **POBDII_ERROR_NO_MEMORY**: Failed to allocate a buffer to store the expected responses.

**Remarks:** User should always check the error code of the generic response to assert that the response data is valid.

**Example**

The following example shows the use of the function OBDII_RequestTestResults on the channel POBDII_USBBUS1 with the OBDMID 01. Depending on the result, a message will be shown to the user.

**Note:** It is assumed that the channel was already initialized.

**C++**

```cpp
TPOBDIStatus result;
BYTE bufferLength = 8;
TPOBDIIMonitorData buffer[8];

// Send OBDII Service $06 request with OBDMID 01
result = OBDII_RequestTestResults(POBDII_USBBUS1, 0x01, buffer, bufferLength);
if (result == POBDII_ERROR_OK) {
    MessageBox(NULL, "Request received responses.", "Success", MB_OK);
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++) {
        if (buffer[j].RESPONSE.ERRORNR == POBDII_R_NOT_USED) {
            // do something
        }
    }
```
// unused response, stop
break;
}
else {
    // process response
}
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox(NULL, "Request received no response.", "Success", MB_OK);
else
    // An error occurred
    MessageBox(NULL, "An error occurred", "Error", MB_OK);

See also: TPOBDIIIMonitorData on page 18

Class-method: RequestTestResults

### 3.7.13 OBDII_RequestPendingTroubleCodes

Sends an OBDII Service “Request Emission-Related Diagnostic Trouble Codes Detected During Current or Last Completed Driving Cycle” (S07) request into queue and waits to receive the responses. The purpose of this service is to enable the external test equipment to obtain “pending” diagnostic trouble codes detected during current or last completed driving cycle for emission-related components/systems.

**Syntax**

#### C++

```cpp
TPOBDIIStatus __stdcall OBDII_RequestPendingTroubleCodes(
    TPOBDIICANHandle CanChannel,
    TPOBDIIDTCData* Data,
    BYTE DataLen);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Obdmid</td>
<td>The On-Board Monitoring Identifier to request</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIDTC structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

**Returns**

The return value is a `TPOBDIIStatus` code. `POBDII_ERROR_OK` is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>POBDII_ERROR_CODE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_ERROR_NOT_INITIALIZED</td>
<td>Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MESSAGE</td>
<td>Indicates that no matching message was received</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MEMORY</td>
<td>Failed to allocate a buffer to store the expected responses</td>
</tr>
</tbody>
</table>

**Remarks**: User should always check the error code of the generic response to assert that the response data is valid.

**Example**

The following example shows the use of the function `OBDII_RequestPendingTroubleCodes` on the channel `POBDII_USBBUS1`. Depending on the result, a message will be shown to the user.
Note: It is assumed that the channel was already initialized.

C++

```c
TPOBDIIStatus result;
BYTE bufferLength = 8;
TPOBDIIDTCData buffer[8];

// Send OBDII Service 07 request
result = OBDII_RequestPendingTroubleCodes(POBDII_USBBUS1, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox(NULL, "Request received responses.", "Success", MB_OK);
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox(NULL, "Request received no response.", "Success", MB_OK);
else
    // An error occurred
    MessageBox(NULL, "An error occurred", "Error", MB_OK);
```

See also: [TPOBDIIDTCData](#) on page 17

**Class-method:** [RequestPendingTroubleCodes](#)

### 3.7.14 OBDII_RequestControlOperation

Sends an OBDII Service “Request Control of On-Board System, Test or Component” ($08) request into queue and waits to receive the responses. The purpose of this service is to enable the external test equipment to control the operation of an on-board system, test or component.

**Syntax**

C++

```c
TPOBDIIStatus __stdcall OBDII_RequestControlOperation(  
    TPOBDIICANHandle CanChannel,  
    TPOBDIITid Tid,  
    TPOBDIIResponse* Response,  
    BYTE DataLen);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Tid</td>
<td>The Test Identifier to request</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIIMonitoringData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>
Returns
The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_ERROR_NOT_INITIALIZED</td>
<td>Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MESSAGE</td>
<td>Indicates that no matching message was received</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MEMORY</td>
<td>Failed to allocate a buffer to store the expected responses</td>
</tr>
</tbody>
</table>

Remarks: User should always check the error code of the generic response to assert that the response data is valid.

Example
The following example shows the use of the function OBDII_RequestControlOperation on the channel POBDII_USBBUS1 with TID 01. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

```cpp
TPOBDIIStatus result;
BYTE bufferLength = 8;
TPOBDIIResponse buffer[8];

// Send OBDII Service 08 request with TID 01
result = OBDII_RequestControlOperation(POBDII_USBBUS1, 0x01, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox(NULL, "Request received responses.", "Success", MB_OK);
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox(NULL, "Request received no response.", "Success", MB_OK);
else
    // An error occurred
    MessageBox(NULL, "An error occurred", "Error", MB_OK);
```

See also: TOBDIIResponse on page 13

Class-method: RequestControlOperation

3.7.15 OBDII_RequestVehicleInformation
Sends an OBDII Service “Request Vehicle Information” ($09) request into queue and waits to receive the responses. The purpose of this service is to enable the external test equipment to request vehicle-specific vehicle information such as Vehicle Identification Number (VIN) and Calibration IDs.
Syntax
C++

```c
TPOBDIIStatus __stdcall OBDII_RequestVehicleInformation(
    TPOBDIICANHandle CanChannel,
    TPOBDIIInfoType InfoType,
    TPOBDIIInfoData* Data,
    BYTE DataLen);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>InfoType</td>
<td>The InfoType Identifier to request</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIIMonitoringData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

- **POBDII_ERROR_NOT_INITIALIZED**: Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application.
- **POBDII_ERROR_NO_MESSAGE**: Indicates that no matching message was received.
- **POBDII_ERROR_NO_MEMORY**: Failed to allocate a buffer to store the expected responses.

Remarks: User should always check the error code of the generic response to assert that the response data is valid.

Example

The following example shows the use of the function OBDII_RequestVehicleInformation on the channel POBDII_USBUSB1 with InfoType 02 (i.e. request Vehicle Identification Number). Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

```c
TPOBDIIStatus result;
BYTE bufferLength = 8;
TPOBDIIInfoData buffer[8];

// Send OBDII Service $09 request with InfoType 02
result = OBDII_RequestVehicleInformation(POBDII_USBUSB1, 0x02, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox(NULL, "Request received responses.", "Success", MB_OK);
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
```
// process response
}
}
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox(NULL, "Request received no response.", "Success", MB_OK);
else
    // An error occurred
    MessageBox(NULL, "An error occurred", "Error", MB_OK);

See also: TPOBDIIInfoData on page 20

Class-method: RequestVehicleInformation

3.7.16 OBDII_RequestPermanentTroubleCodes

Sends an OBDII Service “Request Emission-Related Diagnostic Trouble Codes with Permanent Status” ($0A) request into queue and waits to receive the responses. The purpose of this service is to enable the external test equipment to obtain all DTCs with “permanent DTC” status.

Syntax

C++

TPOBDIStatus __stdcall OBDII_RequestPermanentTroubleCodes(
    TPOBDIICANHandle CanChannel,
    TPOBDIIDTCData* Data,
    BYTE DataLen);

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanChannel</td>
<td>The handle of a POBDII Channel (see TPOBDIICANHandle on page 23)</td>
</tr>
<tr>
<td>Data</td>
<td>An array of TPOBDIIDTCData structure to store the OBDII responses</td>
</tr>
<tr>
<td>DataLen</td>
<td>Number of elements that can be stored in the Data buffer</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPOBDIStatus code. POBDII_ERROR_OK is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>POBDII_ERROR_NOT_INITIALIZED</th>
<th>Indicates that the given POBDII channel was not found in the list of initialized channels of the calling application</th>
</tr>
</thead>
<tbody>
<tr>
<td>POBDII_ERROR_NO_MESSAGE</td>
<td>Indicates that no matching message was received</td>
</tr>
<tr>
<td>POBDII_ERROR_NO_MEMORY</td>
<td>Failed to allocate a buffer to store the expected responses</td>
</tr>
</tbody>
</table>

Remarks: User should always check the error code of the generic response to assert that the response data is valid.

Example

The following example shows the use of the function OBDII_RequestPermanentTroubleCodes on the channel POBDII_USBBUS1. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C++

TPOBDIStatus result;
BYTE bufferLength = 8;
TPOBDIIDTCData buffer[8];

// Send OBDII Service $0A
result = OBDII_RequestPermanentTroubleCodes(POBDII_USBUSB1, buffer, bufferLength);
if (result == POBDII_ERROR_OK)
{
    MessageBox(NULL, "Request received responses.", "Success", MB_OK);
    // search and remove unused responses
    for (int j = 0; j < bufferLength; j++)
    {
        if (buffer[j].RESPONSE.ERRORNR == POBDII_R_NOT_USED)
        {
            // unused response, stop
            break;
        }
        else
        {
            // process response
        }
    }
}
else if (result == POBDII_ERROR_NO_MESSAGE)
    MessageBox(NULL, "Request received no response.", "Success", MB_OK);
else
    // An error occurred
    MessageBox(NULL, "An error occurred", "Error", MB_OK);

See also: TPOBDIIDTCData on page 17

Class-method: RequestPermanentTroubleCodes
3.8 Definitions

The PCAN-Basic API defines the following values.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN-OBD-2Handle Definitions</td>
<td>Defines the handles for the different PCAN channels</td>
</tr>
<tr>
<td>Parameter Value Definitions</td>
<td>Defines the possible values for setting and getting PCAN’s environment information with the functions ODBII_SetValue and ODBII_GetValue</td>
</tr>
</tbody>
</table>

3.8.1 PCAN-OBD-2 Handle Definitions

Defines the handles for the different PCAN buses (Channels) within a class. This values are used as parameter where a `TPOBDIIHANDLE` is needed.

**Default/Undefined handle:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_NONEBUS</code></td>
<td>0x0</td>
<td>Undefined/default value for a PCAN-ISO-TP Channel</td>
</tr>
</tbody>
</table>

**Handles for the ISA Bus (Not Plug & Play):**

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_ISABUS1</code></td>
<td>0x21</td>
<td>PCAN-ISA interface, channel 1</td>
</tr>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_ISABUS2</code></td>
<td>0x22</td>
<td>PCAN-ISA interface, channel 2</td>
</tr>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_ISABUS3</code></td>
<td>0x23</td>
<td>PCAN-ISA interface, channel 3</td>
</tr>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_ISABUS4</code></td>
<td>0x24</td>
<td>PCAN-ISA interface, channel 4</td>
</tr>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_ISABUS5</code></td>
<td>0x25</td>
<td>PCAN-ISA interface, channel 5</td>
</tr>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_ISABUS6</code></td>
<td>0x26</td>
<td>PCAN-ISA interface, channel 6</td>
</tr>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_ISABUS7</code></td>
<td>0x27</td>
<td>PCAN-ISA interface, channel 7</td>
</tr>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_ISABUS8</code></td>
<td>0x28</td>
<td>PCAN-ISA interface, channel 8</td>
</tr>
</tbody>
</table>

**Handles for the Dongle Bus (Not Plug & Play):**

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_DNGBUS1</code></td>
<td>0x31</td>
<td>PCAN-Dongle/LPT interface, channel 1</td>
</tr>
</tbody>
</table>

**Handles for the PCI Bus:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_PCIBUS1</code></td>
<td>0x41</td>
<td>PCAN-PCI interface, channel 1</td>
</tr>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_PCIBUS2</code></td>
<td>0x42</td>
<td>PCAN-PCI interface, channel 2</td>
</tr>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_PCIBUS3</code></td>
<td>0x43</td>
<td>PCAN-PCI interface, channel 3</td>
</tr>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_PCIBUS4</code></td>
<td>0x44</td>
<td>PCAN-PCI interface, channel 4</td>
</tr>
<tr>
<td><code>TPOBDIIHANDLE</code></td>
<td><code>POBDII_PCIBUS5</code></td>
<td>0x45</td>
<td>PCAN-PCI interface, channel 5</td>
</tr>
</tbody>
</table>
Handles for the USB Bus:

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPOBDIICANHandle</td>
<td>PODBII_USBBUS1</td>
<td>0x51</td>
<td>PCAN-USB interface, channel 1</td>
</tr>
<tr>
<td>TPOBDIICANHandle</td>
<td>PODBII_USBBUS2</td>
<td>0x52</td>
<td>PCAN-USB interface, channel 2</td>
</tr>
<tr>
<td>TPOBDIICANHandle</td>
<td>PODBII_USBBUS3</td>
<td>0x53</td>
<td>PCAN-USB interface, channel 3</td>
</tr>
<tr>
<td>TPOBDIICANHandle</td>
<td>PODBII_USBBUS4</td>
<td>0x54</td>
<td>PCAN-USB interface, channel 4</td>
</tr>
<tr>
<td>TPOBDIICANHandle</td>
<td>PODBII_USBBUS5</td>
<td>0x55</td>
<td>PCAN-USB interface, channel 5</td>
</tr>
<tr>
<td>TPOBDIICANHandle</td>
<td>PODBII_USBBUS6</td>
<td>0x56</td>
<td>PCAN-USB interface, channel 6</td>
</tr>
<tr>
<td>TPOBDIICANHandle</td>
<td>PODBII_USBBUS7</td>
<td>0x57</td>
<td>PCAN-USB interface, channel 7</td>
</tr>
<tr>
<td>TPOBDIICANHandle</td>
<td>PODBII_USBBUS8</td>
<td>0x58</td>
<td>PCAN-USB interface, channel 8</td>
</tr>
</tbody>
</table>

Handles for the PC_Card Bus:

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPOBDIICANHandle</td>
<td>PODBII_PCCBUS1</td>
<td>0x61</td>
<td>PCAN-PC Card interface, channel 1</td>
</tr>
<tr>
<td>TPOBDIICANHandle</td>
<td>PODBII_PCCBUS2</td>
<td>0x62</td>
<td>PCAN-PC Card interface, channel 2</td>
</tr>
</tbody>
</table>

**Note**: These definitions are constants values in an object oriented environment (Delphi, .NET Framework) and declared as defines in C++ and Pascal (plain API).

**Hardware Type and Channels:**

**Not Plug & Play**: The hardware channels of this kind are used as registered. This mean, for example, it is allowed to register the PODBII_ISABUS3 without having registered PODBII_ISA1 and PODBII_ISA2. It is a decision of each user, how to associate a POBDII-Channel (logical part) and a port/interrupt pair (physical part).

**Plug & Play**: For hardware handles of PCI, USB and PC-Card, the availability of the channels is determined by the count of hardware connected to a computer in a given moment, in conjunction with their internal handle. This means that having four PCAN-USB connected to a computer will let the user to connect the channels PODBII_USBBUS1 to PODBII_USBBUS4. The association of each channel with hardware is managed internally using the handle of hardware.

**See also**: Parameter Value Definitions below.

### 3.8.2 Parameter Value Definitions

Defines the possible values for setting and getting PCAN-OBD-2 environment information with the functions PODBII_SetValue and PODBII_GetValue.
Debug-Configuration Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>POBDII_DEBUG_NONE</td>
<td>0</td>
<td>No CAN debug messages are being generated</td>
</tr>
<tr>
<td>Int32</td>
<td>POBDII_DEBUG_CAN</td>
<td>1</td>
<td>CAN debug messages are written to the stdout output</td>
</tr>
</tbody>
</table>

Channel Availability Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>POBDII_CHANNEL_UNAVAILABLE</td>
<td>0</td>
<td>The OBDII PCAN-Channel handle is illegal, or its associated hardware is not available</td>
</tr>
<tr>
<td>Int32</td>
<td>POBDII_CHANNEL_AVAILABLE</td>
<td>1</td>
<td>The OBDII PCAN-Channel handle is valid to connect/initialize. Furthermore, for plug&amp;play hardware, this means that the hardware is plugged-in</td>
</tr>
<tr>
<td>Int32</td>
<td>POBDII_CHANNEL_OCCUPIED</td>
<td>2</td>
<td>The OBDII PCAN-Channel handle is valid, and is currently being used</td>
</tr>
</tbody>
</table>

Logging-Configuration Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>POBDII_LOGGING_NONE</td>
<td>0</td>
<td>Disable logging</td>
</tr>
<tr>
<td>Int32</td>
<td>POBDII_LOGGING_TO_FILE</td>
<td>1</td>
<td>Log to a file (log file is automatically with the current local time)</td>
</tr>
<tr>
<td>Int32</td>
<td>POBDII_LOGGING_TO_STDOUT</td>
<td>2</td>
<td>Log to standard output</td>
</tr>
</tbody>
</table>

Baudrate-Configuration Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>POBDII_BAUDRATE_NON_LEGISLATED</td>
<td>0</td>
<td>CAN BUS is initialized with a non legislated-OBDII baudrate. Note this is used only when returned by OBDII_GetValue function with parameter POBDII_PARAM_BAUDRATE</td>
</tr>
<tr>
<td>Int32</td>
<td>POBDII_BAUDRATE_250K</td>
<td>1</td>
<td>250 kbit/s</td>
</tr>
<tr>
<td>Int32</td>
<td>POBDII_BAUDRATE_500K</td>
<td>2</td>
<td>500 kbit/s</td>
</tr>
<tr>
<td>Int32</td>
<td>POBDII_BAUDRATE_AUTODETECT</td>
<td>255</td>
<td>Use autodetection to detect baudrate (used for initialization only)</td>
</tr>
</tbody>
</table>

CAN Identifier Length Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>POBDII_CAN_ID_11BIT</td>
<td>11</td>
<td>11 Bit CAN Identifier length</td>
</tr>
<tr>
<td>Int32</td>
<td>POBDII_CAN_ID_29BIT</td>
<td>29</td>
<td>29 Bit CAN Identifier length</td>
</tr>
</tbody>
</table>

Note: These definitions are constants values in an object oriented environment (Delphi, .NET Framework) and declared as defines in C++ and Pascal (plain API).

See also: TPOBDIIParameter on page 31, PCAN-OBD-2 Handle Definitions on page 123
4 Additional Information

PCAN is the platform for PCAN-OBD-2, PCAN-UDS and PCAN-Basic. In the following topics there is an overview of PCAN and the fundamental practice with the interface DLL CanApi2 (PCAN-API).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN Fundamentals</td>
<td>This section contains an introduction to PCAN</td>
</tr>
<tr>
<td>PCAN-Basic</td>
<td>This section contains general information about the PCAN-Basic API</td>
</tr>
<tr>
<td>UDS und ISO-TP network addressing</td>
<td>This section contains general information about the ISO-TP network addressing</td>
</tr>
</tbody>
</table>

4.1 PCAN Fundamentals

PCAN is a synonym for PEAK CAN APPLICATIONS and is a flexible system for planning, developing, and using a CAN Bus System. Developers as well as end users are getting a helpful and powerful product.

Basis for the communication between PCs and external hardware via CAN is a series of Windows Kernel Mode Drivers (Virtual Device Drivers) e.g. PCAN_USB.SYS, PCAN_PCI.SYS, PCAN_xxx.SYS. These drivers are the core of a complete CAN environment on a PC running Windows and work as interfaces between CAN software and PC-based CAN hardware. The drivers manage the entire data flow of every CAN device connected to the PC.

A user or administrator of a CAN installation gets access via the PCAN-Clients (short: Clients). Several parameters of processes can be visualized and changed with their help. The drivers allow the connection of several Clients at the same time.

Furthermore, several hardware components based on the SJA1000 CAN controller are supported by a PCAN driver. So-called Nets provide the logical structure for CAN busses, which are virtually extended into the PC. On the hardware side, several Clients can be connected, too. The following figures demonstrate different possibilities of Net configurations (also realizable at the same time):

Following rules apply to PCAN clients, nets and hardware:

- One Client can be connected to several Nets
- One Net provides several Clients
- One piece of hardware belongs to one Net
- One Net can include none or one piece of hardware
- A message from a transmitting Client is carried on to every other connected Client, and to the external bus via the connected CAN hardware
- A message received by the CAN hardware is received by every connected Client. However, Clients react only on those messages that pass their acceptance filter
Users of PCAN-View 3 do not have to define and manage Nets. If PCAN-View is instructed to connect directly to PCAN hardware, the application automatically creates a Net for the selected hardware, and automatically establishes a connection with this Net.

See also: PCAN-Basic below, ISO-TP Network Addressing Format on page 129

4.2 PCAN-Basic

PCAN-Basic is an Application Programming Interface for the use of a collection of Windows Device Drivers from PEAK-System, which allow the real-time connection of Windows applications to all CAN busses physically connected to a PC.

PCAN-Basic principal characteristics are:

- Information about the receive time of a CAN message
- Easy switching between different PCAN-Channels (PCAN-PC hardware)
- The possibility to control some parameters in the hardware, e.g. "Listen-Only" mode, automatic reset of the CAN controller, etc.
- The use of event notifications, for faster processing of incoming CAN messages
- An improved system for debugging operations
- The use of only one Dynamic Link Library (PCANBasic.DLL) for all supported hardware
- The possibility to connect more than 2 channels per PCAN-Device. The following list shows the PCAN-Channels that can be connected per PCAN-Device:

<table>
<thead>
<tr>
<th></th>
<th>PCAN-ISA</th>
<th>PCAN-Dongle</th>
<th>PCAN-PCI</th>
<th>PCAN-USB</th>
<th>PCAN-PC-Card</th>
<th>PCAN-LAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of channels</td>
<td>8</td>
<td>1</td>
<td>16</td>
<td>16</td>
<td>2</td>
<td>16</td>
</tr>
</tbody>
</table>

Using the PCAN-Basic

The PCAN-basic offers the possibility to use several PCAN-Channels within the same application in an easy way. The communication process is divided in 3 phases: initialization, interaction and finalization of a PCAN-Channel.

Initialization: In order to do CAN communication using a channel, it is necessary to first initialize it. This is done making a call to the function CAN_Initialize (class-method: Initialize) or CAN_InitializeFD (class-method: InitializeFD) in case FD communication is desired.

Interaction: After a successful initialization, a channel is ready to communicate with the connected CAN bus. Further configuration is not needed. The functions CAN_Read and CAN_Write (class-methods: Read and Write) can be then used to read and write CAN messages. If the channel being used is FD capable and it was initialized using CAN_InitializedFD, then the functions to use are CAN_ReadFD and CAN_WriteFD (class-methods: ReadFD and WriteFD). If desired, extra configuration can be made to improve a communication session, like changing the message filter to target specific messages.

Finalization: When the communication is finished, the function CAN_Uninitialize (class-method: Uninitialize) should be called in order to release the PCAN-Channel and the resources allocated for it. In this way the channel is marked as "Free" and can be used from other applications.

Hardware and Drivers

Overview of the current PCAN hardware and device drivers:
## Hardware

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Plug-and-Play Hardware</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN-Dongle</td>
<td>No</td>
<td>Pcan_dng.sys</td>
</tr>
<tr>
<td>PCAN-ISA</td>
<td>No</td>
<td>Pcan_isa.sys</td>
</tr>
<tr>
<td>PCAN-PC/104</td>
<td>No</td>
<td>Pcan_isa.sys</td>
</tr>
<tr>
<td>PCAN-PCI</td>
<td>Yes</td>
<td>Pcan_pci.sys</td>
</tr>
<tr>
<td>PCAN-PCI Express</td>
<td>Yes</td>
<td>Pcan_pci.sys</td>
</tr>
<tr>
<td>PCAN-cPCI</td>
<td>Yes</td>
<td>Pcan_pci.sys</td>
</tr>
<tr>
<td>PCAN-miniPCI</td>
<td>Yes</td>
<td>Pcan_pci.sys</td>
</tr>
<tr>
<td>PCAN-PC/104-Plus</td>
<td>Yes</td>
<td>Pcan_pci.sys</td>
</tr>
<tr>
<td>PCAN-USB</td>
<td>Yes</td>
<td>Pcan_usb.sys</td>
</tr>
<tr>
<td>PCAN-USB Pro</td>
<td>Yes</td>
<td>Pcan_usb.sys</td>
</tr>
<tr>
<td>PCAN-USB Pro FD</td>
<td>Yes</td>
<td>Pcan_usb.sys</td>
</tr>
<tr>
<td>PCAN-PC Card</td>
<td>Yes</td>
<td>Pcan_pcc.sys</td>
</tr>
<tr>
<td>PCAN-Ethernet Gateway DR</td>
<td>Yes</td>
<td>Pcan_lan.sys</td>
</tr>
<tr>
<td>PCAN-Wireless Gateway DR</td>
<td>Yes</td>
<td>Pcan_lan.sys</td>
</tr>
<tr>
<td>PCAN- Wireless Gateway</td>
<td>Yes</td>
<td>Pcan_lan.sys</td>
</tr>
<tr>
<td>PCAN- Wireless Automotive Gateway DR</td>
<td>Yes</td>
<td>Pcan_lan.sys</td>
</tr>
</tbody>
</table>

See also: PCAN Fundamentals on page 126, ISO-TP Network Addressing Format on page 129.
4.3 OBDII, UDS, and ISO-TP Network Addressing Information

The PCAN-OBD-2 API is built on top of the UDS/ISO-TP API, the following configuration is automatically set when the API is loaded in order to do legislated OBD-communication:

- Only the normal addressing format is used in the case of 11 bit CAN identifiers
- Only the normal fixed addressing format is used in the case of 29 bit CAN identifiers
- ISO-TP blocksize parameter is defined to 0,
- ISO-TP separationTime parameter is defined to 0,
- ISO-TP WaitForTransmission parameter is defined to 0
- Since UDS API is already configured to allow legislated-OBD communication, no extra configuration is made (see UDS and ISO-TP Network Addressing Information)

4.3.1 UDS and ISO-TP Network Addressing Information

The UDS API makes use of the PCAN-ISO-TP API to receive and transmit UDS messages. When a PCAN-UDS Channel is initialized, the ISO-TP API is configured to allow the following communications:

- Functional request using 11 bits CAN identifier and normal addressing, from External Test Equipment address (PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT) to OBD functional address (PUDS_ISO_15765_4_ADDR_OBD_FUNCTIONAL),
- Physical requests and responses using 11 bits CAN identifier and normal addressing, between the External Test Equipment address (PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT) and standard ECU addresses (ECU #1 to #8)
- Communications with 29 bits CAN identifier and FIXED NORMAL addressing format
- Communications with 29 bits CAN identifier and MIXED addressing format
- Communications with 29 bits CAN identifier and ENHANCED addressing format

If an application requires other communication settings, it will have to be set with through the PCAN-ISO-TP API. Although PCAN-UDS and PCAN-ISO-TP define different types for CAN channels (respectively TPUDSCANHandle and TPCANTPHandle), they are both the same type. Once a PCAN-UDS channel is initialized, PCAN-ISO-TP specific functions (like CANTP_AddMapping) can be called with this PCAN-UDS channel.

4.3.2 ISO-TP Network Addressing Format

ISO-TP specifies three addressing formats to exchange data: normal, extended and mixed addressing. Each addressing requires a different number of CAN frame data bytes to encapsulate the addressing information associated with the data to be exchanged.

The following table sums up the mandatory configuration to the ISO-TP API for each addressing format:

<table>
<thead>
<tr>
<th>Addressing Format</th>
<th>CAN ID Length</th>
<th>Mandatory Configuration Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal addressing</td>
<td>11 bits</td>
<td>Define mappings with CANTP_AddMapping</td>
</tr>
<tr>
<td>PCANTP_FORMAT_NORMAL</td>
<td>29 bits</td>
<td>Define mappings with CANTP_AddMapping</td>
</tr>
<tr>
<td>Normal fixed addressing</td>
<td>11 bits</td>
<td>Addressing is invalid</td>
</tr>
<tr>
<td>PCANTP_FORMAT_FIXED_NORMAL</td>
<td>29 bits</td>
<td>-</td>
</tr>
<tr>
<td>Extended addressing</td>
<td>11 bits</td>
<td>Define mappings with CANTP_AddMapping</td>
</tr>
<tr>
<td>PCANTP_FORMAT_EXTENDED</td>
<td>29 bits</td>
<td>Define mappings with CANTP_AddMapping</td>
</tr>
<tr>
<td>Mixed addressing</td>
<td>11 bits</td>
<td>Define mappings with CANTP_AddMapping</td>
</tr>
<tr>
<td>PCANTP_FORMAT_MIXED</td>
<td>29 bits</td>
<td>-</td>
</tr>
</tbody>
</table>
A mapping allows an ISO-TP node to identify and decode CAN Identifiers, it binds a CAN ID to an ISO-TP network address information. CAN messages that cannot be identified are ignored by the API.

Mappings involving physically addressed communication are most usually defined in pairs: the first mapping defines outgoing communication (i.e. request messages from node A to node B) and the second to match incoming communication (i.e. responses from node B to node A).

Functionally addressed communication requires one mapping to transmit functionally addressed messages (i.e. request messages from node A to any node) and as many mappings as responding nodes.