Welcome to the documentation of PCAN-Basic, the new small Version of the PCAN-API from PEAK-System.

- [Introduction](#)  - [DLL API Reference](#)  - [Additional Information](#)

_Last Update: 31.07.2017_
Introduction

Welcome to the documentation to PCAN-Basic.

The PCAN system of the company PEAK-System Technik GmbH consists of a collection of Windows Device Drivers, which allow the real-time connection of Windows applications to all CAN busses physically connected to a PC.

PCAN-Basic, successor of PCAN-Light, is a simple programming interface to the PCAN system. Via the PCAN-Basic Dll it is possible to connect own applications to the Device drivers and the PCAN hardware, to communicate with the CAN busses.

In this Chapter

<table>
<thead>
<tr>
<th>Topics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding PCAN-Basic</td>
<td>This section contains an introduction to PCAN-Basic API.</td>
</tr>
<tr>
<td>Using Events</td>
<td>Offers support about how to read CAN messages using notifications.</td>
</tr>
<tr>
<td>License Regulations</td>
<td>License regulations to this software.</td>
</tr>
<tr>
<td>Contact information</td>
<td>Contact information - PEAK-System Technik GmbH.</td>
</tr>
</tbody>
</table>
PCAN-Basic is the new version of the PCAN-Light API. It consists of a collection of Windows Device Drivers which allow the real-time connection of Windows applications to all CAN busses physically connected to a PC.

Main differences between PCAN-Light and PCAN-Basic

- 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, eg. "Listen-Only" mode, automatic reset of the CAN controller on bus-off, etc.
- The use of event notifications, for faster processing of incoming CAN messages.
- An improved system for debugging's operations.
- The use of only one Dynamic Link Library (PCAN-Basic.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 CANInitializedFD, 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:

<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>Device</td>
<td>Supported</td>
<td>Driver</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------</td>
<td>----------</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 FD</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</td>
<td>Yes</td>
<td>Pcan_lan.sys</td>
</tr>
</tbody>
</table>

See Also

- PCAN Fundamentals
- PCAN-Light
- PCAN-API
Event objects can be used to automatically notify a client on reception of a CAN message. This has following advantages:

- The client program doesn't need to check periodically for received messages any longer.
- The response time on received messages is reduced.

To use events, the client application must call the `CAN_SetValue` (class-method: `SetValue`) function to set the parameter `PCAN_RECEIVE_EVENT`. This parameter sets the handle for the event object. When receiving a message, the driver sets this event to the "Signaled" state.

Another thread must be started in the client application, which waits for the event to be signaled, using one of the Win32 synchronization functions (e.g. `WaitForSingleObject`) without increasing the processor load. After the event is signaled, the receive buffer of the client can be read with the `CAN_Read` (class method: `Read`) function, and the CAN messages can be processed.

Remarks

Tips for the creation of the event object:

- Creation of the event as "auto-reset"
  - Trigger mode "set" (default): After the first waiting thread has been released, the event object's state changes to non-signaled. Other waiting threads are not released. If no threads are waiting, the event object's state remains signaled.
  - Trigger mode "pulse": After the first waiting thread has been released, the event object's state changes to non-signaled. Other waiting threads are not released. If no threads are waiting, or if no thread can be released immediately, the event object's state is simply set to non-signaled.
- Creation of the event as "manual-reset"
- Trigger mode "set" (default): The state of the event object remains signaled until it is set explicitly to the non-signaled state by the Win32 \texttt{ResetEvent} function. Any number of waiting threads, or threads that subsequently begin wait operations, can be released while the object's state remains signaled.

- Trigger mode "pulse": All waiting threads that can be released immediately are released. The event object's state is then reset to the non-signaled state. If no threads are waiting, or if no thread can be released immediately, the event object's state is simply set to non-signaled.

\section*{See Also}

- \texttt{CAN\_SetValue} (class-method: \texttt{SetValue})
- \texttt{CAN\_Read} (class-method: \texttt{Read})
Namings for products in this manual, that are registered trademarks, are not separately marked. Therefore the missing of the ® sign does not implicate, that the naming is a free trade name. Furthermore the used names do not indicate patent rights or anything similar. PEAK-System Technik GmbH makes no representation or warranties with respect to the use of enclosed software or the contents of this manual, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, PEAK-System Technik GmbH reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Copyright © 2000-2015 PEAK-System Technik GmbH
All rights reserved.

No part of this publication may be reproduced, photocopied, stored on a retrieval system, or transmitted without the express written consent of PEAK-System Technik GmbH.

See Also

Contact Information
PCAN-Basic Documentation

Contact Information

This software is a product of:

![PEAK System Logo]

PEAK-System Technik GmbH
Otto-Röhm-Str. 69
64293 Darmstadt, Germany

<table>
<thead>
<tr>
<th>Info:</th>
<th><a href="mailto:info@peak-system.com">info@peak-system.com</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Support:</td>
<td><a href="mailto:support@peak-system.com">support@peak-system.com</a></td>
</tr>
<tr>
<td>Web:</td>
<td><a href="http://www.peak-system.com">www.peak-system.com</a></td>
</tr>
</tbody>
</table>

Copyright © 2017. PEAK-System Technik GmbH. All rights reserved.
Send feedback to this documentation
This section contains information about the data types (classes, structures, types, defines, enumerations) and API functions which are contained in the PCAN-Basic API.

In this Chapter

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Namespaces</td>
<td>Lists the defined namespaces for Microsoft's .NET Framework programming environment.</td>
</tr>
<tr>
<td>Modules</td>
<td>Lists the defined modules for Python 2.6 programming environment.</td>
</tr>
<tr>
<td>Units</td>
<td>Lists the defined units for Delphi's programming environment.</td>
</tr>
<tr>
<td>Classes</td>
<td>Lists the defined classes that implement the PCAN-Basic API.</td>
</tr>
<tr>
<td>Structures</td>
<td>Lists the defined structures.</td>
</tr>
<tr>
<td>Types</td>
<td>Lists the defined types.</td>
</tr>
<tr>
<td>Methods</td>
<td>Lists the defined class methods for using the PCAN-Basic API.</td>
</tr>
<tr>
<td>Functions</td>
<td>List the defined functions for using the PCAN-Basic API.</td>
</tr>
<tr>
<td>Definitions</td>
<td>Lists the defined values.</td>
</tr>
</tbody>
</table>
**PEAK** offers the implementation of some specific programming interfaces as namespaces for the .NET Framework programming environment. The following namespaces are available to be used:

<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.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>
The Peak.Can.Basic namespace contain types and classes for using the PCAN-Basic API within the .NET Framework programming environment, in order to handle with PCAN devices from PEAK-System.

Remarks

Under the Delphi environment, these elements are enclosed in the PCANBasic-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).

Aliases

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCANHandle</td>
<td>Represents a PCAN-hardware channel handle.</td>
</tr>
<tr>
<td>TPCANBitrateFD</td>
<td>Represents a bit rate with flexible data rate.</td>
</tr>
<tr>
<td>TPCANTimestampFD</td>
<td>Represents the timestamp of a CAN message with flexible data rate.</td>
</tr>
</tbody>
</table>

Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCANBasic</td>
<td>Defines a class which represents the PCAN-Basic API.</td>
</tr>
</tbody>
</table>

Structures
### Structure

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCANMsg</td>
<td>Defines a CAN message.</td>
</tr>
<tr>
<td>TPCANTimestamp</td>
<td>Defines a time-stamp of a CAN message.</td>
</tr>
<tr>
<td>TPCANMsgFD</td>
<td>Defines a CAN message with flexible data rate.</td>
</tr>
</tbody>
</table>

### Enumerations

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCANStatus</td>
<td>Represents a PCAN status/error code.</td>
</tr>
<tr>
<td>TPCANDevice</td>
<td>Represents a PCAN device.</td>
</tr>
<tr>
<td>TPCANParameter</td>
<td>Represents a PCAN parameter to be read or set.</td>
</tr>
<tr>
<td>TPCANMessageType</td>
<td>Represents the type of a CAN message</td>
</tr>
<tr>
<td>TPCANType</td>
<td>Represents the type of a Not-Plug-And-Play PCAN hardware.</td>
</tr>
<tr>
<td>TPCANMode</td>
<td>Represents a PCAN filter mode.</td>
</tr>
<tr>
<td>TPCANBaudrate</td>
<td>Represents a PCAN bit rate register value.</td>
</tr>
</tbody>
</table>
PEAK offers the implementation of some specific programming interfaces as modules for programming under Python (ver. 2.6). The following modules are available to be used:

### Modules

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN-Basic-Module</td>
<td>Python module for using the PCAN-Basic API from PEAK-System.</td>
</tr>
</tbody>
</table>
The Peak.Can.Basic module contains types and classes for using the PCAN-Basic API within Python 2.6 programming environment, in order to handle with PCAN devices from PEAK-System.

## Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCANBasic</td>
<td>Defines a class which represents the PCAN-Basic API.</td>
</tr>
</tbody>
</table>

## Structures

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCANMsg</td>
<td>Defines a CAN message. The members of this structure are sequentially byte aligned.</td>
</tr>
<tr>
<td>TPCANTimestamp</td>
<td>Defines a time-stamp of a CAN message.</td>
</tr>
<tr>
<td>TPCANMsgFD</td>
<td>Defines a CAN message with flexible data rate.</td>
</tr>
</tbody>
</table>
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:

### Units

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

The PCANBasic-Unit contain types and classes for using the PCAN-Basic API within the Delphi's programming environment, in order to handle with PCAN devices from PEAK-System.

Remarks

For the .NET Framework, these elements are enclosed in the Peak.Can.Basic 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>TPCANHandle</td>
<td>Represents a PCAN-hardware channel handle.</td>
</tr>
<tr>
<td>TPCANBitrateFD</td>
<td>Represents a bit rate with flexible data rate.</td>
</tr>
<tr>
<td>TPCANTimestampFD</td>
<td>Represents the timestamp of a CAN message with flexible data rate.</td>
</tr>
</tbody>
</table>

Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCANBasic</td>
<td>Defines a class which represents the PCAN-Basic API.</td>
</tr>
</tbody>
</table>

Structures
## Structure

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCANMsg</td>
<td>Defines a CAN message.</td>
</tr>
<tr>
<td>TPCANTimestamp</td>
<td>Defines a time-stamp of a CAN message.</td>
</tr>
<tr>
<td>TPCANMsgFD</td>
<td>Defines a CAN message with flexible data rate.</td>
</tr>
</tbody>
</table>

## Enumerations

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCANStatus</td>
<td>Represents a PCAN status/error code.</td>
</tr>
<tr>
<td>TPCANDevice</td>
<td>Represents a PCAN device.</td>
</tr>
<tr>
<td>TPCANParameter</td>
<td>Represents a PCAN parameter to be read or set.</td>
</tr>
<tr>
<td>TPCANMessageType</td>
<td>Represents the type of a CAN message</td>
</tr>
<tr>
<td>TPCANType</td>
<td>Represents the type of a Not-Plug-And-Play PCAN hardware.</td>
</tr>
<tr>
<td>TPCANMode</td>
<td>Represents a PCAN filter mode.</td>
</tr>
<tr>
<td>TPCANBaudrate</td>
<td>Represents a PCAN bit rate register value.</td>
</tr>
</tbody>
</table>
The following classes are offered to make use of the PCAN-Basic API in a managed or unmanaged way.

### Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCANBasic</td>
<td>Defines a class to use the PCAN-Basic API within the Microsoft’s .NET Framework programming environment, and Python.</td>
</tr>
<tr>
<td>TPCANBasic</td>
<td>Defines a class to use the PCAN-Basic API within the Delphi programming environment.</td>
</tr>
</tbody>
</table>
PCANBasic

Defines a class which represents the PCAN-Basic API for using within the Microsoft's .NET Framework and Python (ver. 2.6).

Syntax

- **C#**
  ```csharp
  public static class PCANBasic
  ```

- **C++ / CLR**
  ```c++
  public ref class PCANBasic abstract sealed
  ```

- **Visual Basic**
  ```vbnet
  Public NotInheritable Class PCANBasic
  ```

- **Python**
  ```python
  class PCANBasic
  ```

Remarks

The PCANBasic class collects and implements the PCAN-Basic API functions. Each method is called just like the API function with the exception that the prefix "CAN_" is not used. The structure and functionality of the methods and API functions is the same.

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

```csharp
TPCANStatus res;
// Static use, without any instance
//
res = PCANBasic.Initialize(PCAN_USBBUS1,PCAN_BAUD_500K);
```

Within Python, a variable must be instantiated with an object of type
PCANBasic, in order to use the API.

```plaintext
# Object instantiation
#
objPCAN = PCANBasic()
res = objPCAN.Initialize(PCAN_USB_BUS1, PCAN_BAUD_500K)
```

**Note** that this class under Delphi is called [TPCANBasic](#).

### See Also

- [Methods](#)
- [Definitions](#)

**Delphi:** [TPCANBasic](#)
Definitions a class which represents the PCAN-Basic API in the Delphi programming environment.

Syntax

<table>
<thead>
<tr>
<th>Pascal OO</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCANBasic = class</td>
</tr>
</tbody>
</table>

Remarks

TPCANBasic is a class containing only class-methods and constant members, allowing their use without the creation of any object, just like an static class of another programming languages. It collects and implements the PCAN-Basic API functions. Each method is called just like the API function with the exception that the prefix "CAN_" is not used. The structure and functionality of the methods and API functions is the same.

Note that this class under .NET Framework is called PCANBasic.

See Also

Methods
Definitions

.NET Framework: PCANBasic
The PCAN-Basic API defines the following structures:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCANMsg</td>
<td>Defines a CAN message.</td>
</tr>
<tr>
<td>TPCANTimestamp</td>
<td>Defines the point of time at which a CAN message was received.</td>
</tr>
<tr>
<td>TPCANMsgFD</td>
<td>Defines a CAN message with flexible data rate.</td>
</tr>
</tbody>
</table>

See Also

Reference
TPCANMsg

Defines a CAN message.

Syntax

```cpp
typedef struct
{
    DWORD ID;
    TPCANMessageType MSGTYPE;
    BYTE LEN;
    BYTE DATA[8];
} TPCANMsg;
```

```pascal
TPCANMsg = record
    ID: Longword;
    MSGTYPE: TPCANMessageType;
    LEN: Byte;
    DATA: array[0..7] of Byte;
end;
```

```csharp
public struct TPCANMsg
{
    public uint ID;
    [MarshalAs(UnmanagedType.U1)]
    public TPCANMessageType MSGTYPE;
    public byte LEN;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 8)]
    public byte[] DATA;
}
```
C++ / CLR

```c++
public value struct TPCANMsg
{
    UInt32 ID;
    [MarshalAs(UnmanagedType.U1)]
    TPCANMessageType MSGTYPE;
    Byte LEN;
    [MarshalAs(UnmanagedType::ByValArray,
        SizeConst	=	8)]
    array<Byte>^ DATA;
}
```

Visual Basic

```vbnet
Public Structure TPCANMsg
    Public ID As UInt32
    [MarshalAs(UnmanagedType.U1)] _
    Public MSGTYPE As TPCANMessageType
    Public LEN As Byte
    [MarshalAs(UnmanagedType.ByValArray,
        SizeConst:=8)] _
    Public DATA As Byte()
End Structure
```

Python

```python
from ctypes import *

class TPCANMsg (Structure):
    _fields_ = [ ("ID", c_uint),
                ("MSGTYPE", TPCANMessageType),
                ("LEN", c_ubyte),
                ("DATA", c_ubyte * 8) ]
```

Remarks

The members of this structure are sequentially byte aligned.

Fields
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>11/29-bit message identifier.</td>
</tr>
<tr>
<td>MSGTYPE</td>
<td>Type of the message. Bit mask indicating the type of the message. Several message types can be combined.</td>
</tr>
<tr>
<td>LEN</td>
<td>Data Length Code of the message (0..8).</td>
</tr>
<tr>
<td>DATA</td>
<td>Data of the message (DATA[0]..DATA[7]).</td>
</tr>
</tbody>
</table>

**See Also**

- CAN_Read (class-method: Read)
- CAN_Write (class-method: Write)
- TPCANTimestamp
TPCANTimestamp

Defines a time-stamp of a CAN message. The time-stamp contains the number of microseconds since the start of Windows.

Syntax

- **C++**
  ```cpp
typedef struct
  {
    DWORD  millis;
    WORD   millis_overflow;
    WORD   micros;
  } TPCANTimestamp;
  ```

- **Pascal OO**
  ```pascal
TPCANTimestamp = record
    millis: Longword;
    millis_overflow: Word;
    micros: Word;
end;
  ```

- **C#**
  ```csharp
public struct TPCANTimestamp
{
    public uint  millis;
    public ushort millis_overflow;
    public ushort micros;
}
  ```

- **C++ / CLR**
  ```csharp
public value struct TPCANTimestamp
{
```
```csharp
UInt32 millis;
UInt16 millis_overflow;
UInt16 micros;
}

Visual Basic

Public Structure TPCANTimestamp
    Public millis As UInt32
    Public millis_overflow As UInt16
    Public micros As UInt16
End Structure

Python

from ctypes import *
class TPCANTimestamp (Structure):
    _fields_ = [ ("millis", c_uint),
                ("millis_overflow", c_ushort),
                ("micros", c_ushort) ]

Remarks

The members of this structure are sequentially byte aligned.

Calculation of total of microseconds: micros + 1000 * millis + 0x100000000 * 1000 * millis_overflow

Fields

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>millis</td>
<td>Base-value: milliseconds: 0.. 2^32-1.</td>
</tr>
<tr>
<td>millis_overflow</td>
<td>Roll-arounds of millis.</td>
</tr>
<tr>
<td>micros</td>
<td>Microseconds: 0..999.</td>
</tr>
</tbody>
</table>

See Also
CAN_Write (class-method: Write)

TPCANMsg
TPCANMsgFD

Defines a CAN message with flexible data rate.

Syntax

**C++**

typedef struct
{
    DWORD ID;
    TPCANMessageType MSGTYPE;
    BYTE DLC;
    BYTE DATA[64];
} TPCANMsgFD;

**Pascal OO**

TPCANMsgFD = record
    ID: Longword;
    MSGTYPE: TPCANMessageType;
    DLC: Byte;
    DATA: array[0..63] of Byte;
end;

**C#**

public struct TPCANMsgFD
{
    public uint ID;
    [MarshalAs(UnmanagedType.U1)] public TPCANMessageType MSGTYPE;
    public byte DLC;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 64)] public byte[] DATA;
}
Remarks

The members of this structure are sequentially byte aligned.

Fields
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>11/29-bit message identifier.</td>
</tr>
<tr>
<td>MSGTYPE</td>
<td>Type of the message. Bit mask indicating the type of the message. Several message types can be combined.</td>
</tr>
<tr>
<td>DLC</td>
<td>Data Length Code of the message (0..15).</td>
</tr>
<tr>
<td>DATA</td>
<td>Data of the message (DATA[0]..DATA[63]).</td>
</tr>
</tbody>
</table>

**Remark**

**Longer Data field with CAN FD messages:**

The length of data bytes contained in a CAN message is given by the DATA LENGTH CODE field (DLC). The coding of the DLC within FD messages is different. There are 7 additional codes (from 9 to 15) that allows a FD Messages to transport up to 64 bytes of data. The relationship between DLC and data bytes length is as follow:

<table>
<thead>
<tr>
<th>DLC</th>
<th>Data Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
See Also

**CAN_ReadFD** (class-method: **ReadFD**)

**CAN_WriteFD** (class-method: **WriteFD**)

---

Copyright © 2017. PEAK-System Technik GmbH. All rights reserved.

Send feedback to this documentation
The PCAN-Basic API defines the following types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TPCANHandle</code></td>
<td>Represents a PCAN hardware channel handle.</td>
</tr>
<tr>
<td><code>TPCANStatus</code></td>
<td>Represents a PCAN status/error code.</td>
</tr>
<tr>
<td><code>TPCANDevice</code></td>
<td>Represents a PCAN device.</td>
</tr>
<tr>
<td><code>TPCANParameter</code></td>
<td>Represents a PCAN parameter to be read or set.</td>
</tr>
<tr>
<td><code>TPCANMessageType</code></td>
<td>Represents the type of a PCAN message.</td>
</tr>
<tr>
<td><code>TPCANType</code></td>
<td>Represents the type of PCAN hardware to be initialized.</td>
</tr>
<tr>
<td><code>TPCANMode</code></td>
<td>Represents a PCAN filter mode.</td>
</tr>
<tr>
<td><code>TPCANBaudrate</code></td>
<td>Represents a PCAN bit rate register value.</td>
</tr>
<tr>
<td><code>TPCANBitrateFD</code></td>
<td>Represents a bit rate string with flexible data rate.</td>
</tr>
<tr>
<td><code>TPCANTimestampFD</code></td>
<td>Represents the timestamp of a CAN message with flexible data rate.</td>
</tr>
</tbody>
</table>
PCAN-Basic Documentation

**TPCANHandle**

Represents a PCAN-hardware channel handle.

**Syntax**

```cpp
#define TPCANHandle WORD
```

```pascal
TPCANHandle = Word;
```

```c#
using TPCANHandle = System.UInt16;
```

```cpp CLR
#define TPCANHandle System::UInt16
```

```vb
Imports TPCANHandle = System.UInt16
```

```python
TPCANHandle = c_ushort
```

**Remarks**

**FD capable Hardware:**

Some hardware can transmit using a flexible data rate (FD capable). Although there are no special PCAN-Handles to identify these hardware, it is possible to ask if a hardware is able to communicate using the FD protocol. The PCAN-Basic parameter **PCAN_CHANNEL_FEATURES** allows to investigate whether a hardware is FD capable before being initialized.

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

Aliases are defined in the Peak.Can.Basic 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.Basic Namespace. Otherwise, just use the native type, which in this case is a UInt16.

C#:

```csharp
using System;
using Peak.Can.Basic;
using TPCANHandle = System(UInt16); // Alias's declaration
```

Visual Basic:

```vbnet
Imports System
Imports Peak.Can.Basic
Imports TPCANHandle = System(UInt16) ' Alias declaration
```

See Also

PCAN Handle Definitions
TPCANStatus

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

Syntax

```c
#define TPCANStatus DWORD

#define PCAN_ERROR_OK 0x00000
#define PCAN_ERROR_XMTFULL 0x00001
#define PCAN_ERROR_OVERRUN 0x00002
#define PCAN_ERROR_BUSLIGHT 0x00004
#define PCAN_ERROR_BUSHEAVY 0x00008
#define PCAN_ERROR_BUSWARNING (PCAN_ERROR_BUSWARNING | PCAN_ERROR_BUSHEAVY)
#define PCAN_ERROR_BUSPASSIVE 0x40000
#define PCAN_ERROR_BUSOFF 0x00010
#define PCAN_ERROR_ANYBUSERR (PCAN_ERROR_BUSWARNING | PCAN_ERROR_BUSHEAVY)
#define PCAN_ERROR_QRCVEMPTY 0x00020
#define PCAN_ERROR_QOVERRUN 0x00040
#define PCAN_ERROR_QXMTFULL 0x00080
#define PCAN_ERROR_REGTEST 0x00100
#define PCAN_ERROR_NODRIVER 0x00200
#define PCAN_ERROR_HWINUSE 0x00400
#define PCAN_ERROR_NETINUSE 0x00800
#define PCAN_ERROR_ILLHW 0x01400
#define PCAN_ERROR_ILLNET 0x01800
#define PCAN_ERROR_ILLCLIENT (PCAN_ERROR_ILLHW | PCAN_ERROR_ILLNET)
#define PCAN_ERROR_ILLHANDLE (PCAN_ERROR_ILLHW | PCAN_ERROR_ILLNET)
#define PCAN_ERROR_ILLPARAMTYPE 0x04000
```
```pascal
{$Z4}
TPCANStatus = (  
  PCAN_ERROR_OK = $00000,  
  PCAN_ERROR_XMTFULL = $00001,  
  PCAN_ERROR_OVERRUN = $00002,  
  PCAN_ERROR_BUSLIGHT = $00004,  
  PCAN_ERROR_BUSHALLOW = $00008,  
  PCAN_ERROR_BUSHWARNING = Byte(PCAN_ERROR_BUSLIGHT),  
  PCAN_ERROR_BUSPASSIVE = $00020,  
  PCAN_ERROR_BUSOFF = $00040,  
  PCAN_ERROR_QRCVEMPTY = $00080,  
  PCAN_ERROR_QOVERRUN = $00100,  
  PCAN_ERROR_QXMTFULL = $00120,  
  PCAN_ERROR_REGTEST = $00200,  
  PCAN_ERROR_NODRIVER = $00400,  
  PCAN_ERROR_HWINUSE = $00800,  
  PCAN_ERROR_NETINUSE = $01000,  
  PCAN_ERROR_ILLHW = $01400,  
  PCAN_ERROR_ILLNET = $01800,  
  PCAN_ERROR_ILLCLIENT = $01C00,  
  PCAN_ERROR_ILLHANDLE = Byte(PCAN_ERROR_ILLCLIENT),  
  PCAN_ERROR_ILLPARAMRESOURCE = $02000,  
  PCAN_ERROR_ILLPARAMTYPE = $04000,  
  PCAN_ERROR_ILLPARAMVAL = $08000,  
  PCAN_ERROR_UNKNOWN = $10000,
```
PCAN_ERROR_I LLDATA = $20000,
PCAN_ERROR_CAUTION = $2000000,
PCAN_ERROR_INITIALIZE = $4000000,
PCAN_ERROR_ILLOPERATION = $8000000
);

[C#]

[Flags]
public enum TPCANStatus : uint
{
    PCAN_ERROR_OK = 0x00000,
    PCAN_ERROR_XMTFULL = 0x00001,
    PCAN_ERROR_OVERRUN = 0x00002,
    PCAN_ERROR_BUSLIGHT = 0x00004,
    PCAN_ERROR_BUSHEAVY = 0x00008,
    PCAN_ERROR_BUSWARNING = PCAN_ERROR_BUSHEAVY,
    PCAN_ERROR_BUSPASSIVE = 0x40000,
    PCAN_ERROR_BUSOFF = 0x00010,
    PCAN_ERROR_ANYBUSERR = (PCAN_ERROR_BUSWARNING | PCAN_ERROR_BUSHEAVY),
    PCAN_ERROR_QRCVEMPTY = 0x00020,
    PCAN_ERROR_QOVERRUN = 0x00040,
    PCAN_ERROR_QXMTFULL = 0x00080,
    PCAN_ERROR_REGTEST = 0x00100,
    PCAN_ERROR_NODRIVER = 0x00200,
    PCAN_ERROR_HWINUSE = 0x00400,
    PCAN_ERROR_NETINUSE = 0x00800,
    PCAN_ERROR_ILLHW = 0x01400,
    PCAN_ERROR_ILLNET = 0x01800,
    PCAN_ERROR_ILLCLIENT = 0x01C00,
    PCAN_ERROR_ILLHANDLE = (PCAN_ERROR_ILLHW | PCAN_ERROR_ILLNET | PCAN_ERROR_ILLCLIENT),
    PCAN_ERROR_ILLPARAMTYPE = 0x04000,
    PCAN_ERROR_ILLPARAMVAL = 0x08000,
    PCAN_ERROR_UNKNOWN = 0x10000,
PCAN_ERROR_ILLDATA = 0x20000,
PCAN_ERROR_CAUTION = 0x2000000,
PCAN_ERROR_INITIALIZE = 0x4000000,
PCAN_ERROR_ILLOPERATION = 0x8000000,
}

[C++/CLR]

[Flags]
public enum class TPCANStatus : UInt32
{

    PCAN_ERROR_OK = 0x00000,
    PCAN_ERROR_XMTFULL = 0x00001,
    PCAN_ERROR_OVERRUN = 0x00002,
    PCAN_ERROR_BUSLIGHT = 0x00004,
    PCAN_ERROR_BUSHEAVY = 0x00008,
    PCAN_ERROR_BUSWARNING = PCAN_ERROR_BUSHEAVY,
    PCAN_ERROR_BUSPASSIVE = 0x40000,
    PCAN_ERROR_BUSOFF = 0x00010,
    PCAN_ERROR_ANYBUSERR = (PCAN_ERROR_BUSWARNING | PCAN_ERROR_BUSLIGHT),
    PCAN_ERROR_QRCVEMPTY = 0x00020,
    PCAN_ERROR_QOVERRUN = 0x00040,
    PCAN_ERROR_QXMTFULL = 0x00080,
    PCAN_ERROR_REGTEST = 0x00100,
    PCAN_ERROR_NODRIVER = 0x00200,
    PCAN_ERROR_HWINUSE = 0x00400,
    PCAN_ERROR_NETINUSE = 0x00800,
    PCAN_ERROR_ILLHW = 0x01400,
    PCAN_ERROR_ILLNET = 0x01800,
    PCAN_ERROR_ILLCLIENT = 0x01C00,
    PCAN_ERROR_ILLHANDLE = (PCAN_ERROR_ILLHW | PCAN_ERROR_ILLCLIENT),
    PCAN_ERROR_ILLRESOURCE = 0x02000,
    PCAN_ERROR_ILLPARAMTYPE = 0x04000,
    PCAN_ERROR_ILLPARAMVAL = 0x08000,
    PCAN_ERROR_UNKNOWN = 0x10000,
}
PCAN_ERROR_ILLDATA = 0x20000,
PCAN_ERROR_CAUTION = 0x2000000,
PCAN_ERROR_INITIALIZE = 0x4000000,
PCAN_ERROR_ILLOPERATION = 0x8000000,

Visual Basic
<Flags()> _
Public Enum TPCANStatus As UInt32
    PCAN_ERROR_OK = &H0
    PCAN_ERROR_XMTFULL = &H1
    PCAN_ERROR_OVERRUN = &H2
    PCAN_ERROR_BUSLIGHT = &H4
    PCAN_ERROR_BUSHEAVY = &H8
    PCAN_ERROR_BUSWARNING = PCAN_ERROR_BUSHEAVY
    PCAN_ERROR_BUSPASSIVE = &H40000
    PCAN_ERROR_BUSOFF = &H10
    PCAN_ERROR_ANYBUSERR = (PCAN_ERROR_BUSWARNING)
    PCAN_ERROR_QRCVEMPTY = &H20
    PCAN_ERROR_QOVERRUN = &H40
    PCAN_ERROR_QXMTFULL = &H80
    PCAN_ERROR_REGTEST = &H100
    PCAN_ERROR_NODRIVER = &H200
    PCAN_ERROR_HWINUSE = &H400
    PCAN_ERROR_NETINUSE = &H800
    PCAN_ERROR_ILLHW = &H1400
    PCAN_ERROR_ILLNET = &H1800
    PCAN_ERROR_ILLCLIENT = &H1C00
    PCAN_ERROR_ILLHANDLE = (PCAN_ERROR_ILLHW Or
    PCAN_ERRORRESOURCE = &H2000
    PCAN_ERROR_ILLPARAMTYPE = &H4000
    PCAN_ERROR_ILLPARAMVAL = &H8000
    PCAN_ERROR_UNKNOWN = &H10000
    PCAN_ERROR_ILLDATA = &H20000
}
PCAN_ERROR_CAUTION = &H2000000
PCAN_ERROR_INITIALIZE = &H4000000
PCAN_ERROR_ILLOPERATION = &H8000000
End Enum

```python
TPCANStatus = int

PCAN_ERROR_OK = TPCANStatus(0x00000)
PCAN_ERROR_XMTFULL = TPCANStatus(0x00001)
PCAN_ERROR_OVERRUN = TPCANStatus(0x00002)
PCAN_ERROR_BUSLIGHT = TPCANStatus(0x00004)
PCAN_ERROR_BUSHEAVY = TPCANStatus(0x00008)
PCAN_ERROR_BUSWARNING = TPCANStatus(PCAN_ERROR_BUSHEAVY)
PCAN_ERROR_BUSPASSIVE = TPCANStatus(0x40000)
PCAN_ERROR_BUSOFF = TPCANStatus(0x00010)
PCAN_ERROR_ANYBUSERR = TPCANStatus(PCAN_ERROR_BUSWARNING | PCAN_ERROR_BUSLIGHT)
PCAN_ERROR_QRCVEMPTY = TPCANStatus(0x00020)
PCAN_ERROR_QOVERRUN = TPCANStatus(0x00040)
PCAN_ERROR_QXMTFULL = TPCANStatus(0x00080)
PCAN_ERROR_REGTEST = TPCANStatus(0x00100)
PCAN_ERROR_NODRIVER = TPCANStatus(0x00200)
PCAN_ERROR_HWINUSE = TPCANStatus(0x00400)
PCAN_ERROR_NETINUSE = TPCANStatus(0x00800)
PCAN_ERROR_ILLHW = TPCANStatus(0x01400)
PCAN_ERROR_ILLNET = TPCANStatus(0x01800)
PCAN_ERROR_ILLCLIENT = TPCANStatus(0x01C00)
PCAN_ERROR_I LLHANDLE = TPCANStatus(PCAN_ERROR_ILLHW)
PCAN_ERROR_RESOURCE = TPCANStatus(0x02000)
PCAN_ERROR_ILLPARAMTYPE = TPCANStatus(0x04000)
PCAN_ERROR_ILLPARAMVAL = TPCANStatus(0x08000)
PCAN_ERROR_UNKNOWN = TPCANStatus(0x10000)
PCAN_ERROR_ILLDATA = TPCANStatus(0x20000)
PCAN_ERROR_CAUTION = TPCANStatus(0x2000000)
```


Remarks

Note that the values of the different PCAN-Status definitions are able to be bitwise combined. In some cases it is possible to get more than one error code as result of calling a function.

*Note* that the values of `PCAN_ERROR_INITIALIZE` and `PCAN_ERROR_ILLOPERATION` were changed!

- `PCAN_ERROR_INITIALIZE` changed from 0x40000 to 0x40000000
- `PCAN_ERROR_ILLOPERATION` changed from 0x80000 to 0x80000000

Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_OK</td>
<td>0x000000</td>
<td>No error. Success.</td>
</tr>
<tr>
<td>PCAN_ERROR_XMTFULL</td>
<td>0x000001</td>
<td>Transmit buffer in CAN controller is full.</td>
</tr>
<tr>
<td>PCAN_ERROR_OVERRUN</td>
<td>0x000002</td>
<td>CAN controller was read too late.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSLIGHT</td>
<td>0x000004</td>
<td>Bus error: an error counter reached the 'light' limit.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSHEAVY</td>
<td>0x000008</td>
<td>Bus error: an</td>
</tr>
<tr>
<td>Error Code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSWARNING</td>
<td>0x00008 (000008)</td>
<td>Bus error: an error counter reached the 'warning' limit.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSPASSIVE</td>
<td>0x40000 (262144)</td>
<td>Bus error: the CAN controller is error passive.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSOFF</td>
<td>0x00010 (000016)</td>
<td>Bus error: the CAN controller is in bus-off state.</td>
</tr>
<tr>
<td>PCAN_ERROR_ANYBUSERR</td>
<td>0x4001C (262172)</td>
<td>Mask for all bus errors.</td>
</tr>
<tr>
<td>PCAN_ERROR_QRCVEMPTY</td>
<td>0x00020 (000032)</td>
<td>Receive queue is empty.</td>
</tr>
<tr>
<td>PCAN_ERROR_QOVERRUN</td>
<td>0x00040 (000064)</td>
<td>Receive queue was read too late.</td>
</tr>
<tr>
<td>PCAN_ERROR_QXMTFULL</td>
<td>0x00080 (000128)</td>
<td>Transmit queue is full.</td>
</tr>
<tr>
<td>PCAN_ERROR_REGTEST</td>
<td>0x00100 (000256)</td>
<td>Test of the CAN controller hardware registers failed (no description).</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>PCAN_ERROR_NODRIVER</td>
<td>0x00200 (000512)</td>
<td>Driver not loaded.</td>
</tr>
<tr>
<td>PCAN_ERROR_HWINUSE</td>
<td>0x00400 (001024)</td>
<td>PCAN-Hardware already in use by a PCAN-Net.</td>
</tr>
<tr>
<td>PCAN_ERROR_NETINUSE</td>
<td>0x00800 (002048)</td>
<td>A PCAN-Client is already connected to the PCAN-Net.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLHW</td>
<td>0x01400 (005120)</td>
<td>PCAN-Hardware handle is invalid.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLNET</td>
<td>0x01800 (006144)</td>
<td>PCAN-Net handle is invalid.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLCLIENT</td>
<td>0x01C00 (007168)</td>
<td>PCAN-Client handle is invalid.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLHANDLE</td>
<td>0x01C00 (007168)</td>
<td>Mask for all handle errors.</td>
</tr>
<tr>
<td>PCAN_ERROR_RESOURCE</td>
<td>0x02000 (008192)</td>
<td>Resource (FIFO, Client, timeout) cannot be found).</td>
</tr>
<tr>
<td>Code</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLPARAMTYPE</td>
<td>0x04000 (016384)</td>
<td>Invalid parameter.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLPARAMVAL</td>
<td>0x08000 (032768)</td>
<td>Invalid parameter value.</td>
</tr>
<tr>
<td>PCAN_ERROR_UNKNOWN</td>
<td>0x10000 (065536)</td>
<td>Unknown error</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLDATA</td>
<td>0x20000 (131072)</td>
<td>Invalid data, function, or action.</td>
</tr>
<tr>
<td>PCAN_ERROR_CAUTION</td>
<td>0x2000000 (33554432)</td>
<td>Operation succeeded but with irregularities.</td>
</tr>
<tr>
<td>PCAN_ERROR_INITIALIZE*</td>
<td>0x4000000 (67108864)</td>
<td>Channel is not initialized.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLOPERATION*</td>
<td>0x8000000 (134217728)</td>
<td>Invalid operation.</td>
</tr>
</tbody>
</table>
TPCANDevice

Represents a PCAN device. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

### C++

```cpp
#define TPCANDevice BYTE

#define PCAN_NONE   0x00
#define PCAN_PEAKCAN 0x01
#define PCAN_ISA     0x02
#define PCAN_DNG     0x03
#define PCAN_PCI     0x04
#define PCAN_USB     0x05
#define PCAN_PCC     0x06
#define PCAN_VIRTUAL 0x07
#define PCAN_LAN     0x08
```

### Pascal OO

```pascal
{$Z1}
TPCANDevice = (  
  PCAN_NONE = 0,  
  PCAN_PEAKCAN = 1,  
  PCAN_ISA = 2,  
  PCAN_DNG = 3,  
  PCAN_PCI = 4,  
  PCAN_USB = 5,  
  PCAN_PCC = 6,  
  PCAN_VIRTUAL = 7,  
  PCAN_LAN = 8
```
public enum TPCANDevice : byte
{
    PCAN_NONE = 0,
    PCAN_PEAKCAN = 1,
    PCAN_ISA = 2,
    PCAN_DNG = 3,
    PCAN_PCI = 4,
    PCAN_USB = 5,
    PCAN_PCC = 6,
    PCAN_VIRTUAL = 7,
    PCAN_LAN = 8
}

public enum class TPCANDevice : Byte
{
    PCAN_NONE = 0,
    PCAN_PEAKCAN = 1,
    PCAN_ISA = 2,
    PCAN_DNG = 3,
    PCAN_PCI = 4,
    PCAN_USB = 5,
    PCAN_PCC = 6,
    PCAN_VIRTUAL = 7,
    PCAN_LAN = 8
};

Public Enum TPCANDevice As Byte
    PCAN_NONE = 0
    PCAN_PEAKCAN = 1
PCAN_ISA = 2
PCAN_DNG = 3
PCAN_PCI = 4
PCAN_USB = 5
PCAN_PCC = 6
PCAN_VIRTUAL = 7
PCAN_LAN = 8

End Enum

Python

TPCANDevice = c_ubyte

PCAN_NONE = TPCANDevice(0x00)
PCAN_PEAKCAN = TPCANDevice(0x01)
PCAN_ISA = TPCANDevice(0x02)
PCAN_DNG = TPCANDevice(0x03)
PCAN_PCI = TPCANDevice(0x04)
PCAN_USB = TPCANDevice(0x05)
PCAN_PCC = TPCANDevice(0x06)
PCAN_VIRTUAL = TPCANDevice(0x07)
PCAN_LAN = TPCANDevice(0x08)

Remarks

The PCAN-Devices PCAN_PEAKCAN and PCAN_VIRTUAL are not used within the PCAN-Basic API.

Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_NONE</td>
<td>0</td>
<td>Undefined, unknown or not selected PCAN device value.</td>
</tr>
<tr>
<td>PCAN_PEAKCAN</td>
<td>1</td>
<td>PCAN Non-Plug And Play devices. NOT USED WITHIN PCAN-Basic</td>
</tr>
<tr>
<td>Device</td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>PCAN_ISA</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PCAN_DNG</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PCAN_PCI</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PCAN_USB</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>PCAN_PCC</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>PCAN_VIRTUAL</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>PCAN_LAN</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

- PCAN-ISA, PCAN-PC/104.
- PCAN-Dongle.
- PCAN-PCI, PCAN-cPCI, PCAN-miniPCI, PCAN-PC/104-Plus, and PCAN-PCI Express.
- PCAN-USB and PCAN-USB Pro.
- PCAN-PC Card.
- PCAN Virtual hardware. NOT USED WITHIN PCAN-Basic API.
- PCAN Gateway devices.
TPCANParameter

Represents a PCAN parameter or a PCAN Value that can 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++

```c
#define TPCANParameter BYTE
#define PCAN_DEVICE_NUMBER 0x01
#define PCAN_5VOLTS_POWER 0x02
#define PCAN_RECEIVE_EVENT 0x03
#define PCAN_MESSAGE_FILTER 0x04
#define PCAN_API_VERSION 0x05
#define PCAN_CHANNEL_VERSION 0x06
#define PCAN_BUSOFF_AUTORESET 0x07
#define PCAN_LISTEN_ONLY 0x08
#define PCAN_LOG_LOCATION 0x09
#define PCAN_LOG_STATUS 0x0A
#define PCAN_LOG_CONFIGURE 0x0B
#define PCAN_LOG_TEXT 0x0C
#define PCAN_CHANNEL_CONDITION 0x0D
#define PCAN_HARDWARE_NAME 0x0E
#define PCAN_RECEIVE_STATUS 0x0F
#define PCAN_CONTROLLER_NUMBER 0x10
#define PCAN_TRACE_LOCATION 0x11
#define PCAN_TRACE_STATUS 0x12
#define PCAN_TRACE_SIZE 0x13
#define PCAN_TRACE_CONFIGURE 0x14
```
```c
#define PCAN_CHANNEL_IDENTIFYING 0x15
#define PCAN_CHANNEL_FEATURES 0x16
#define PCAN_BITRATE_ADAPTING 0x17
#define PCAN_BITRATE_INFO 0x18
#define PCAN_BITRATE_INFO_FD 0x19
#define PCAN_BUSSPEED_NOMINAL 0x1A
#define PCAN_BUSSPEED_DATA 0x1B
#define PCAN_IP_ADDRESS 0x1C
#define PCAN_LAN_SERVICE_STATUS 0x1D
#define PCAN_ALLOW_STATUS_FRAMES 0x1E
#define PCAN_ALLOW_RTR_FRAMES 0x1F
#define PCAN_ALLOW_ERROR_FRAMES 0x20
#define PCAN_INTERFRAME_DELAY 0x21
#define PCAN_ACCEPTANCE_FILTER_11BIT 0x22
#define PCAN_ACCEPTANCE_FILTER_29BIT 0x23

Pascal OO

{$Z1}$
TPCANParameter = ( 
    PCAN_DEVICE_NUMBER = 1,
    PCAN_5VOLTS_POWER = 2,
    PCAN_RECEIVE_EVENT = 3,
    PCAN_MESSAGE_FILTER = 4,
    PCAN_API_VERSION = 5,
    PCAN_CHANNEL_VERSION = 6,
    PCAN_BUSOFF_AUTORESET = 7,
    PCAN_LISTEN_ONLY = 8,
    PCAN_LOG_LOCATION = 9,
    PCAN_LOG_STATUS = 10,
    PCAN_LOG_CONFIGURE = 11,
    PCAN_LOG_TEXT = 12,
    PCAN_CHANNEL_CONDITION = 13,
    PCAN_HARDWARE_NAME = 14,
    PCAN_RECEIVE_STATUS = 15,
)
PCAN_CONTROLLER_NUMBER = 16,
PCAN_TRACE_LOCATION = 17,
PCAN_TRACE_STATUS = 18,
PCAN_TRACE_SIZE = 19,
PCAN_TRACE_CONFIGURE = 20,
PCAN_CHANNEL_IDENTIFYING = 21,
PCAN_CHANNEL_FEATURES = 22,
PCAN_BITRATE_ADAPTING = 23,
PCAN_BITRATE_INFO = 24,
PCAN_BITRATE_INFO_FD = 25,
PCAN_BUSSPEED_NOMINAL = 26,
PCAN_BUSSPEED_DATA = 27,
PCAN_IP_ADDRESS = 28,
PCAN_LAN_SERVICE_STATUS = 29,
PCAN_ALLOW_STATUS_FRAMES = 30,
PCAN_ALLOW_RTR_FRAMES = 31,
PCAN_ALLOW_ERROR_FRAMES = 32,
PCAN_INTERFRAME_DELAY = 33,
PCAN_ACCEPTANCE_FILTER_11BIT = 34,
PCAN_ACCEPTANCE_FILTER_29BIT = 35

public enum TPCANParameter : byte
{
    PCAN_DEVICE_NUMBER = 1,
    PCAN_5VOLTS_POWER = 2,
    PCAN_RECEIVE_EVENT = 3,
    PCAN_MESSAGE_FILTER = 4,
    PCAN_API_VERSION = 5,
    PCAN_CHANNEL_VERSION = 6,
    PCAN_BUSOFF_AUTORESET = 7,
    PCAN_LISTEN_ONLY = 8,
    PCAN_LOG_LOCATION = 9,
PCAN_LOG_STATUS = 10,
PCAN_LOG_CONFIGURE = 11,
PCAN_LOG_TEXT = 12,
PCAN_CHANNEL_CONDITION = 13,
PCAN_HARDWARE_NAME = 14,
PCAN_RECEIVE_STATUS = 15,
PCAN_CONTROLLER_NUMBER = 16,
PCAN_TRACE_LOCATION = 17,
PCAN_TRACE_STATUS = 18,
PCAN_TRACE_SIZE = 19,
PCAN_TRACE_CONFIGURE = 20,
PCAN_CHANNEL_IDENTIFYING = 21,
PCAN_CHANNEL_FEATURES = 22,
PCAN_BITRATE_ADAPTING = 23,
PCAN_BITRATE_INFO = 24,
PCAN_BITRATE_INFO_FD = 25,
PCAN_BUSSPEED_NOMINAL = 26,
PCAN_BUSSPEED_DATA = 27,
PCAN_IP_ADDRESS = 28,
PCAN_LAN_SERVICE_STATUS = 29,
PCAN_ALLOW_STATUS_FRAMES = 30,
PCAN_ALLOW_RTR_FRAMES = 31,
PCAN_ALLOW_ERROR_FRAMES = 32,
PCAN_INTERFRAME_DELAY = 33,
PCAN_ACCEPTANCE_FILTER_11BIT = 34,
PCAN_ACCEPTANCE_FILTER_29BIT = 35,

C++ / CLR

public enum class TPCANParameter : Byte {
    PCAN_DEVICE_NUMBER = 1,
    PCAN_5VOLTS_POWER = 2,
    PCAN_RECEIVE_EVENT = 3,
}
Public Enum TPCANParameter As Byte
    PCAN_DEVICE_NUMBER = 1
    PCAN_5VOLTS_POWER = 2
    PCAN_RECEIVE_EVENT = 3
    PCAN_MESSAGE_FILTER = 4
    PCAN_API_VERSION = 5
    PCAN_CHANNEL_VERSION = 6
    PCAN_BUSOFF_AUTORESET = 7
    PCAN_LISTEN_ONLY = 8
    PCAN_LOG_LOCATION = 9
    PCAN_LOG_STATUS = 10
    PCAN_LOG_CONFIGURE = 11
    PCAN_LOG_TEXT = 12
    PCAN_CHANNEL_CONDITION = 13
    PCAN_HARDWARE_NAME = 14
    PCAN_RECEIVE_STATUS = 15
    PCAN_CONTROLLER_NUMBER = 16
    PCAN_TRACE_LOCATION = 17
    PCAN_TRACE_STATUS = 18
    PCAN_TRACE_SIZE = 19
    PCAN_TRACE_CONFIGURE = 20
    PCAN_CHANNEL_IDENTIFYING = 21
    PCAN_CHANNEL_FEATURES = 22
    PCAN_BITRATE_ADAPTING = 23
    PCAN_BITRATE_INFO = 24
    PCAN_BITRATE_INFO_FD = 25
    PCAN_BUSSPEED_NOMINAL = 26
    PCAN_BUSSPEED_DATA = 27
    PCAN_IP_ADDRESS = 28
    PCAN_LAN_SERVICE_STATUS = 29
    PCAN_ALLOW_STATUS_FRAMES = 30
    PCAN_ALLOW_RTR_FRAMES = 31
    PCAN_ALLOW_ERROR_FRAMES = 32
    PCAN_INTERFRAME_DELAY = 33
PCAN_ACCEPTANCE_FILTER_11BIT = 34
PCAN_ACCEPTANCE_FILTER_29BIT = 35

End Enum

**Python**

```python
TPCANParameter = c_ubyte

PCAN_DEVICE_NUMBER = TPCANParameter(0x01)
PCAN_5VOLTS_POWER = TPCANParameter(0x02)
PCAN_RECEIVE_EVENT = TPCANParameter(0x03)
PCAN_MESSAGE_FILTER = TPCANParameter(0x04)
PCAN_API_VERSION = TPCANParameter(0x05)
PCAN_CHANNEL_VERSION = TPCANParameter(0x06)
PCAN_BUSOFF_AUTORESET = TPCANParameter(0x07)
PCAN_LISTEN_ONLY = TPCANParameter(0x08)
PCAN_LOG_LOCATION = TPCANParameter(0x09)
PCAN_LOG_STATUS = TPCANParameter(0x0A)
PCAN_LOG_CONFIGURE = TPCANParameter(0x0B)
PCAN_LOG_TEXT = TPCANParameter(0x0C)
PCAN_CHANNEL_CONDITION = TPCANParameter(0x0D)
PCAN_HARDWARE_NAME = TPCANParameter(0x0E)
PCAN_RECEIVE_STATUS = TPCANParameter(0x0F)
PCAN_CONTROLLER_NUMBER = TPCANParameter(0x10)
PCAN_TRACE_LOCATION = TPCANParameter(0x11)
PCAN_TRACE_STATUS = TPCANParameter(0x12)
PCAN_TRACE_SIZE = TPCANParameter(0x13)
PCAN_TRACE_CONFIGURE = TPCANParameter(0x14)
PCAN_CHANNEL_IDENTIFYING = TPCANParameter(0x15)
PCAN_CHANNEL_FEATURES = TPCANParameter(0x16)
PCAN_BITRATE_ADAPTING = TPCANParameter(0x17)
PCAN_BITRATE_INFO = TPCANParameter(0x18)
PCAN_BITRATE_INFO_FD = TPCANParameter(0x19)
PCAN_BUSSPEED_NOMINAL = TPCANParameter(0x1A)
PCAN_BUSSPEED_DATA = TPCANParameter(0x1B)
```
PCAN_IP_ADDRESS = TPCANParameter(0x1C)
PCAN_LAN_SERVICE_STATUS = TPCANParameter(0x1D)
PCAN_ALLOW_STATUS_FRAMES = TPCANParameter(0x1E)
PCAN_ALLOW_RTR_FRAMES = TPCANParameter(0x1F)
PCAN_ALLOW_ERROR_FRAMES = TPCANParameter(0x20)
PCAN_INTERFRAME_DELAY = TPCANParameter(0x21)
PCAN_ACCEPTANCE_FILTER_11BIT = TPCANParameter(0x22)
PCAN_ACCEPTANCE_FILTER_29BIT = TPCANParameter(0x23)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_DEVICE_NUMBER</td>
<td>1</td>
<td>Integer</td>
<td>PCAN-USB &quot;device number&quot; parameter.</td>
</tr>
<tr>
<td>PCAN_5VOLTS_POWER</td>
<td>2</td>
<td>Integer</td>
<td>PCAN-PC Card &quot;5-volt power&quot; parameter.</td>
</tr>
<tr>
<td>PCAN_RECEIVE_EVENT</td>
<td>3</td>
<td>Handle</td>
<td>PCAN receive event handler parameter.</td>
</tr>
<tr>
<td>PCAN_MESSAGE_FILTER</td>
<td>4</td>
<td>Integer</td>
<td>PCAN message filter parameter.</td>
</tr>
<tr>
<td>PCAN_API_VERSION</td>
<td>5</td>
<td>String</td>
<td>PCAN-Basic API version parameter.</td>
</tr>
<tr>
<td>PCAN_CHANNEL_VERSION</td>
<td>6</td>
<td>String</td>
<td>PCAN device channel version parameter.</td>
</tr>
<tr>
<td>PCAN_BUSOFF_AUTORESET</td>
<td>7</td>
<td>Integer</td>
<td>PCAN &quot;reset on bus-off&quot; parameter.</td>
</tr>
<tr>
<td>PCAN_LISTEN_ONLY</td>
<td>8</td>
<td>Integer</td>
<td>PCAN &quot;listen-only&quot; parameter.</td>
</tr>
<tr>
<td>PCAN_LOG_LOCATION</td>
<td>9</td>
<td>String</td>
<td>Directory path for log parameter.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Value</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------</td>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PCAN_LOG_STATUS</td>
<td>10</td>
<td>Integer</td>
<td>Debug-Log activation status.</td>
</tr>
<tr>
<td>PCAN_LOG_CONFIGURE</td>
<td>11</td>
<td>Integer</td>
<td>Configuration of the debugged information (LOG_FUNCTION_***).</td>
</tr>
<tr>
<td>PCAN_LOG_TEXT</td>
<td>12</td>
<td>String</td>
<td>Custom insertion of text into the log file.</td>
</tr>
<tr>
<td>PCAN_CHANNEL_CONDITION</td>
<td>13</td>
<td>Integer</td>
<td>Availability status of a PCAN-Channel.</td>
</tr>
<tr>
<td>PCAN_HARDWARE_NAME</td>
<td>14</td>
<td>String</td>
<td>PCAN &quot;hardware name&quot; parameter.</td>
</tr>
<tr>
<td>PCAN_RECEIVE_STATUS</td>
<td>15</td>
<td>Integer</td>
<td>&quot;Receive Status&quot; parameter for incoming messages.</td>
</tr>
<tr>
<td>PCAN_CONTROLLER_NUMBER</td>
<td>16</td>
<td>Integer</td>
<td>Index of a CAN-Controller in a PCAN device.</td>
</tr>
<tr>
<td>PCAN_TRACE_LOCATION</td>
<td>17</td>
<td>String</td>
<td>Directory path for PCAN trace files.</td>
</tr>
<tr>
<td>PCAN_TRACE_STATUS</td>
<td>18</td>
<td>Integer</td>
<td>PCAN-Trace activation status.</td>
</tr>
<tr>
<td>PCAN_TRACE_SIZE</td>
<td>19</td>
<td>Integer</td>
<td>Configuration of the maximum size for a PCAN-Trace.</td>
</tr>
<tr>
<td>PCAN_TRACE_CONFIGURE</td>
<td>20</td>
<td>Integer</td>
<td>Configuration of the trace file storing modes (TRACE_FILE_***).</td>
</tr>
<tr>
<td>PCAN_CHANNEL_IDENTIFYING</td>
<td>21</td>
<td>Integer</td>
<td>USB Channel Identifying activation.</td>
</tr>
<tr>
<td>Variable</td>
<td>ID</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----</td>
<td>--------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>PCAN_CHANNEL_FEATURES</td>
<td>22</td>
<td>Integer</td>
<td>Capabilities of a PCAN device (FEATURE_***).</td>
</tr>
<tr>
<td>PCAN_BITRATE_ADAPTING</td>
<td>23</td>
<td>Integer</td>
<td>Attachment to an existing connection with unknown/different bit rate.</td>
</tr>
<tr>
<td>PCAN_BITRATE_INFO</td>
<td>24</td>
<td>Integer</td>
<td>Current bit rate as BTR0BTR1 value (Standard CAN).</td>
</tr>
<tr>
<td>PCAN_BITRATE_INFO_FD</td>
<td>25</td>
<td>String</td>
<td>Current bit rate as String value (CAN-FD).</td>
</tr>
<tr>
<td>PCAN_BUSSPEED_nominal</td>
<td>26</td>
<td>Integer</td>
<td>Current nominal CAN bus speed in bits/second.</td>
</tr>
<tr>
<td>PCAN_BUSSPEED_DATA</td>
<td>27</td>
<td>Integer</td>
<td>Current CAN data speed in bits/second.</td>
</tr>
<tr>
<td>PCAN_IP_ADDRESS</td>
<td>28</td>
<td>String</td>
<td>Remote address as a IPv4 formatted string.</td>
</tr>
<tr>
<td>PCAN_LAN_SERVICE_STATUS</td>
<td>29</td>
<td>Integer</td>
<td>Running status of the LAN Service (Virtual PCAN-Gateway).</td>
</tr>
<tr>
<td>PCAN.Allow_STATUS FRAMES</td>
<td>30</td>
<td>Integer</td>
<td>&quot;Receive Status&quot; parameter for Status frames.</td>
</tr>
<tr>
<td>PCAN_ALLOW_RTR_FRAMES</td>
<td>31</td>
<td>Integer</td>
<td>&quot;Receive Status&quot; parameter for RTR frames.</td>
</tr>
<tr>
<td>PCAN.Allow_ERROR_FRAMES</td>
<td>32</td>
<td>Integer</td>
<td>&quot;Receive Status&quot; parameter for Error frames.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Access</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>PCAN_INTERFRAME_DELAY</td>
<td></td>
<td>Delay, in microseconds, between frames.</td>
<td></td>
</tr>
<tr>
<td>PCAN_ACCEPTANCE_FILTER_11BIT</td>
<td></td>
<td>64-Bit Integer Acceptance filter over code and mask for 11-bit CAN IDs.</td>
<td></td>
</tr>
<tr>
<td>PCAN_ACCEPTANCE_FILTER_29BIT</td>
<td></td>
<td>64-Bit Integer Acceptance filter over code and mask for 29-bit CAN IDs.</td>
<td></td>
</tr>
</tbody>
</table>

## Characteristics

### PCAN_DEVICE_NUMBER

**Access:** **RW**

**Description:** This parameter is used on PCAN-USB hardware to distinguish between 2 (or more) of them on the same computer. This value is persistent, i.e. the identifier will not be lost after disconnecting and connecting again the hardware.

**Possible values:** According with the Firmware version, this value can be a number in the range [1..255] or [1..4294967295]. If the Firmware has a resolution of one byte and the specified value is bigger, than the value is truncated.

**Default value:** If this parameter was never set before, the value is the maximum value possible for the used resolution. For 8-bits: 255 (FFh), for 32 bits: 429496729 (FFFFFFFFh).

**PCAN-Device:** PCAN-USB.

### PCAN_5VOLTS_POWER

**Access:** **RW**

**Description:** This parameter is used on PCAN-PC Card hardware for...
switching the external 5V on the D-Sub connector of the PC Card. This is useful when connecting external bus converter modules to the card (AU5790 / TJA1054)).

**Possible values**: This parameter can have one of these values: \text{PCAN\_PARAMETER\_OFF, PCAN\_PARAMETER\_ON}. Note that other values will considered invalid.

**Default value**: Disabled (PCAN\_PARAMETER\_OFF).

**PCAN-Device**: PCAN-PC Card, PCAN-HUB.

---

**PCAN\_RECEIVE\_EVENT**

**Access**: \text{RW}

**Description**: This parameter is used to let the PCAN driver notify an application when CAN messages are placed in its receive queue. In this form, message processing tasks of an application can react faster and make a more efficient use of the processor time.

**Possible values**: This value has to be a handle for an event object returned by the Windows API function \text{CreateEvent} or the value \text{0} (\text{IntPtr\_Zero} in a managed environment). When setting this parameter, the value of \text{0} resets the parameter in the PCAN driver. When reading the value of \text{0} indicate that no event handle is set. For more information about reading with events, please refer to the topic \text{Using Events}.

**Default value**: Disabled (0).

**PCAN-Device**: All PCAN devices (excluding PCAN\_NONEBUS channel).

---

**PCAN\_MESSAGE\_FILTER**

**Access**: \text{RW}

**Description**: This parameter allows the user to easy configure the message filter of a PCAN channel. With it is possible to fully open or complete close the filter.
**Possible values:** When setting only two values are possible: **PCAN_FILTER_OPEN, PCAN_FILTER_CLOSE**. When reading it is possible to receive a third value, **PCAN_FILTER_CUSTOM**, which indicates that the filter is configured to receive a custom range of IDs. Note that other values will considered invalid.

**Default value:** Complete opened (PCAN_FILTER_OPEN).

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

---

**PCAN_API_VERSION**

**Access:** ❇️

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

**Possible values:** The value is a null-terminated string indication the version number of the API implementation. The returned text has the following form: \textit{\texttt{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 PCAN_NONEBUS channel.

---

**PCAN_CHANNEL_VERSION**

**Access:** ❇️

**Description:** This parameter is used to get version information about the Driver of a PCAN Channel.

**Possible values:** The value is a null-terminated string which contains version number, driver name and copyright information about the driver used to handle with an specified PCAN channel. The length of
the this text will have a maximum length of 256 bytes (null-termination included).

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 version.

PCAN_BUSOFF_AUTORESET

Access: RW

Description: This parameter instructs the PCAN driver to reset automatically the CAN controller of a PCAN channel when a bus-off state is detected. Since no communication is possible on a bus-off state, it is useful to let the driver to catch this event automatically and reset the controller, avoiding extra handling of this problem in an end application.

Possible values: This parameter can have one of these values: PCAN_PARAMETER_OFF, PCAN_PARAMETER_ON. Note that other values will considered invalid.

Default value: Disabled (PCAN_PARAMETER_OFF).

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

REMARKS: Reseting the hardware has a duration of ~ 500 milliseconds. After receiving the PCAN_ERROR_BUSOFF error, an application should wait that time before trying to read or write again.

PCAN_LISTEN_ONLY

Access: RW

Description: This parameter allows the user to set a CAN hardware in Listen-Only mode. When this mode is set, the CAN controller does't take part on active events (eg. transmit CAN messages) but stays in a passive mode (CAN monitor), in which it can analyse the
traffic on the CAN bus used by a PCAN channel. See also the Philips Data Sheet "SJA1000 Stand-alone CAN controller".

**Possible values:** This parameter can have one of these values: **PCAN_PARAMETER_OFF, PCAN_PARAMETER_ON**. Note that other values will considered invalid.

**Default value:** Disabled (PCAN_PARAMETER_OFF).

**PCAN-Device:** All PCAN devices (excluding PCAN_NONEBUS) containing a SJA1000 CAN controller.

**REMARKS:** This parameter can be used with an initialized or uninitialized channel. Configuring this parameter without having the channel initialized, does a so called "pre-initialization". This means that the channel will be set in Listen-Only mode as soon as possible, after it has been successfully connected, using **CAN_Initialize**(class-method: Initialize). Once the channel is disconnected, further initializations of this channel are done in normal mode. It is needed to set this parameter again before each initialization process, if the behavior described before is required. This is useful to avoid or minimize arbitration problems when connecting to a CAN-network.

---

**PCAN_LOG_LOCATION**

**Access:** **RW**

**Description:** This value is used to set the folder location on a computer for the Log-File generated by the PCAN-Basic API, within a debug session. Setting this value starts recording debug information automatically. If a debug session is running (a log file is being written), **PCAN_LOG_LOCATION** instructs the API to close the current log file and to start the process again with the new folder information. Note that the name of the log file cannot be specified, this name is fixed as PCANBasic.log.

**Possible values:** This value must be a fully-qualified and valid path to an existing directory on the executing computer. There is no limit for the length of the string but it is recommended to use a length not bigger than MAX_PATH. For more information see **Naming Files**,
Paths, and Namespaces.

Default value: Calling process's folder.

PCAN-Device: Default channel Only (PCAN_NONEBUS).

PCAN_LOG_STATUS

Access: RW

Description: This value is used to control the activity status of a debug session within the PCAN-Basic API. If the log status is set to ON without having set a location for the log file or without having configured the information to be traced, then the session process will start with the default values.

Possible values: The value must be one of the following values: PCAN_PARAMETER_OFF, PCAN_PARAMETER_ON. Note that other values will considered invalid.

Default value: Disabled (PCAN_PARAMETER_OFF).

PCAN-Device: Default channel Only (PCAN_NONEBUS).

PCAN_LOG_CONFIGURE

Access: RW

Description: This value is used to configure the debug information to be included in the log file generated in a debug session within the PCAN-Basic API.

Possible values: The value must be one of the following values or a combination of them:

- LOG_FUNCTION_DEFAULT: This value is always active. It defines the default information to be traced, which is an unexpected exception like a memory access violation. After having configured the log with more options, this value can be used to reset that configuration, setting the log with its default value again. This kind of entry is marked with the word "EXCEPTION"* in the log file.
- **LOG_FUNCTION_ENTRY**: This value causes an entry in the log file each time an API function is entered. This kind of entry is marked with the word "ENTRY"* in the log file.

- **LOG_FUNCTION_PARAMETERS**: This value causes an entry in the log file each time an API function is entered, showing the name of the parameters passed to the function as well as their values. This kind of entry is marked with the word "PARAMETERS"* in the log file.

- **LOG_FUNCTION_LEAVE**: This value causes an entry in the log file each time an API function is abandoned. This kind of entry is marked with the word "EXIT"* in the log file.

- **LOG_FUNCTION_WRITE**: This value causes an entry in the log file each time a CAN message is written, using the function **CAN_Write** (class-method: **Write**). This kind of entry is marked with the word "CHANNEL 0xXX (Y)"* in the log file, where XX is the channel number in hex notation, and Y the word "OUT"* denoting the direction (outgoing). The complete CAN message is also represented as hex text.

- **LOG_FUNCTION_READ**: This value causes an entry in the log file each time a CAN message is read, using the functions **CAN_Read/CAN_ReadFD** (class-methods: **Read, ReadFD**). This kind of entry is marked with the word "CHANNEL 0xXX (Y)"* in the log file, where XX is the channel number in hex notation, and Y the word "IN"* denoting the direction (incoming). The complete CAN message is also represented as hex text.

* Note that the PCAN-Basic API supports several languages. The log file use the language of the operating system. If this language is not one of the supported languages, than English is used.

* These words are always written in English, independently of the operating system's language.

**Default value**: Exceptions and Errors (LOG_FUNCTION_DEFAULT).

**PCAN-Device**: Default channel Only (PCAN_NONEBUS).
**Access:** \textsuperscript{W}

**Description:** This value is used to insert custom information in the log file generated in a debug session within the PCAN-Basic API. Setting this value starts recording debug information automatically. This is very useful when it is desired to specially mark places of an application’s execution path while debugging PCAN-Basic tasks. Furthermore, an application could use this feature as an own Log file. To do so, just use the default log's configuration (PCAN_LOG_CONFIGURE set to LOG_FUNCTION_DEFAULT) and include the desired information using PCAN_LOG_TEXT. In this way the log file will contain only user-defined debug information. Note that the name of the log file cannot be specified, this name is fixed as PCANBasic.log.

**Possible values:** This value must be a null-terminated string. There is no limit for the length of the string but it is recommended to use a length not bigger than MAX_PATH. For more information see \ref{Naming Files, Paths, and Namespaces}.

**Default value:** NA.

**PCAN-Device:** Default channel Only (PCAN_NONEBUS).

\begin{verbatim}
PCAN_CHANNEL_CONDITION
\end{verbatim}

**Access:** \textsuperscript{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:
PCAN_CHANNEL_UNAVAILABLE, PCAN_CHANNEL_AVAILABLE, PCAN_CHANNEL_OCCUPIED, PCAN_CHANNEL_PCANVIEW.

**Default value:** N/A.

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

**PCAN_HARDWARE_NAME**  
**Access:** R

**Description:** This parameter is used to retrieve the name of the hardware represented by a PCAN channel. This is useful when an application wants to differentiate between several models of the same device, e.g. a PCAN-USB and a PCAN-USB Pro.

**Possible values:** The value is a null-terminated string which contains the name of the hardware specified by the given PCAN channel. The length of this text will have a maximum length of 32 bytes (null-termination included).

**Default value:** N/A.

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

**REMARKS:** This parameter can be used with an initialized or uninitialized channel.

**PCAN_RECEIVE_STATUS**  
**Access:** RW

**Description:** This parameter helps the user to allow / disallow the reception of messages within a PCAN Channel, regardless of the value of its reception filter. When the "Receive Status" is active (ON), incoming messages are forwarded to the user application through the CAN_Read/CAN_ReadFD functions (class-methods: Read, ReadFD). If "Receive Status" is deactivated (OFF), the incoming messages are disposed from the receive queue and each call to CAN_Read/CAN_ReadFD returns PCAN_ERROR_QRCVEMPTY. The acceptance filter of the channel remains unchanged (other applications working with the same PCAN-Hardware will not be disturbed).

**Possible values:** This parameter can have one of these values:
PCAN_PARAMETER_OFF, PCAN_PARAMETER_ON. Note that other values will considered invalid.

**Default value:** Activated (PCAN_PARAMETER_ON).

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

**REMARKS:** This parameter can be used with an initialized or uninitialized channel. Configuring this parameter without having the channel initialized, does a so called "pre-initialization". This means that the channel will set the configured "Receive Status" after it has been successfully connected, using CAN.Initialize (class-method: Initialize). Once the channel is disconnected, further initializations of this channel are done with the default value of this parameter (ON). This is usefull to avoid receiving messages immediately after connection, or before the receive filter is configured according to the needs of each application.

---

**PCAN_CONTROLLER_NUMBER**

**Access:** R

**Description:** This parameter is a zero-based index used to identify the CAN controllers built in a hardware. This parameter is useful when it is needed to communicate with a specific physical channel on a multichannel CAN Hardware, e.g. "0" or "1" on a PCAN-USB Pro device.

**Possible values:** A number in the range [0..n-1], where n is the number of physical channels on the device being used.

**Default value:** NA.

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

**REMARKS:** This parameter can be used with an initialized or uninitialized channel.

---

**PCAN_TRACE_LOCATION**
Access: **RW**

**Description**: This value is used to set the folder location on a computer for the PCAN-Trace file generated by the PCAN-Basic API, within a trace session. If a trace session is active (a trace file is being written), PCAN_TRACE_LOCATION instructs the API to close the current trace file and to start recording data again with the new folder information.

**Possible values**: This value must be a fully-qualified and valid path to an existing directory on the executing computer. There is no limit for the length of the string but it is recommended to use a length not bigger than MAX_PATH. For more information see [Naming Files, Paths, and Namespaces](#). Passing an empty string ("", NULL value) instructs the API to use the default value for this parameter.

**Default value**: Calling process's folder.

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

**REMARKS**: Note that the name of the trace file cannot be specified. The file uses the name of the current connection (PCAN-Channel's name) plus a file counter (e.g. PCAN_PCIBUS1_1.trc), though it can be enhanced by issuing the parameter PCAN_TRACE_CONFIGURE.

**PCAN_TRACE_STATUS**

Access: **RW**

**Description**: This value is used to control the activity status of a trace session within the PCAN-Basic API. If the trace status is set to ON without having set a location for the trace file or without having configured the storing mode, then the session process will start with the default values. Trying to activate a trace session can fail if overwriting is not set and a file with the same name already exists.

**Possible values**: The value must be one of the following values: PCAN_PARAMETER_OFF, PCAN_PARAMETER_ON. Note that other values will considered invalid.
**Default value:** Disabled (PCAN_PARAMETER_OFF).

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

---

**PCAN_TRACE_SIZE**

**Access:** R W

**Description:** This value is used to set the maximum size, in megabytes, that a single trace file can have. Trying to set the size for a trace file will fail if the trace session is active.

**Possible values:** A number in the range [1..100], representing the amount of megabytes. Passing a value of 0 instructs the API to use the default value for this parameter. Trying to set a size bigger than 100 will fail.

**Default value:** 10 megabytes.

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

---

**PCAN_TRACE_CONFIGURE**

**Access:** R W

**Description:** This value is used to configure the trace process and the file generated in a trace session within the PCAN-Basic API. Trying to configure a trace file will fail if the trace session is active.

**Possible values:** The value must be one of the following values or a combination of them:

- **TRACE_FILE_SINGLE:** This value represents the default trace configuration. It defines the use of a single trace file as output. When the tracing process starts, the file will be filled out with messages until the size of the file reaches the configured maximum size (see **PCAN_TRACE_SIZE**). The tracing process is then automatically stopped.

- **TRACE_FILE_SEGMENTED:** This value indicates the API to
keep tracing information by using several files. When the trace file being used reaches the maximum configured file size (see **PCAN_TRACE_SIZE**), then a new file is automatically created and the tracing process continues.

- **TRACE_FILE_DATE**: This value instruct the API to use the start date information within the name of the trace file. The format used is YYYYMMDD, four digits for year, the next two for the month, and the last two for the day, e.g. "20130228_PCAN_USBBUS_1" for the 28th February 2013. If both, TRACE_FILE_DATE and TRACE_FILE_TIME are configured, the file name starts always with the date: "20130228140733_PCAN_USBBUS1_1.trc".

- **TRACE_FILE_TIME**: This value instruct the API to use the start time information within the name of the trace file. The format used is HHMMSS, two digits for the hour (24 hours format), the next two for the minutes, and the last two for the seconds, e.g. "140733_PCAN_USBBUS_1" for 14:07:33 (02:07:33 PM). If both, TRACE_FILE_DATE and TRACE_FILE_TIME are configured, the file name starts always with the date: "20130228140733_PCAN_USBBUS1_1.trc".

- **TRACE_FILE_OVERWRITE**: This value causes the overwriting of an existing trace file when a new trace session is started. If this value is not configured, trying to start a tracing process will fail, if the file name to generate is the same as one used by an existing file.

**Default value**: TRACE_FILE_SINGLE (Single file, not overwriting, with standard name).

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

**PCAN_CHANNEL_IDENTIFYING**

**Access**: 

**Description**: This value is used to control the status of the "channel identifying procedure" on USB devices within the PCAN-Basic API. The procedure consists in blinking the LED associated to the given
Possible values: The value must be one of the following values: PCAN_PARAMETER_OFF, PCAN_PARAMETER_ON. Note that other values will considered invalid.

Default value: Disabled (PCAN_PARAMETER_OFF).

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

REMARKS: This parameter can be used with an initialized or uninitialized channel. This identifying procedure is only available for USB based hardware (PCAN-USB, PCAN-USB Pro and PCAN-USB Hub). The blinking of the LED can be different according to the kind of hardware used (in color and blink rate). Only one channel can blink simultaneously.

PCAN_CHANNEL_FEATURES

Access: R

Description: This value is used to read the particularities of a PCAN Channel.

Possible values: The value can be one of the following values or a combination of them:

- FEATURE_FD_CAPABLE: This value indicates that the hardware represented by a PCAN Channel is FD capable (it supports flexible data rate).
- FEATURE_DELAY_CAPABLE: This value indicates that the hardware represented by a PCAN Channel allows the configuration of a delay between sending frames.

Default value: A value of 0, indicating "no special features".

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

REMARKS: This parameter can be used with an initialized or uninitialized channel. FD Hardware must be initialized with CAN_InitializeFD (class-method: InitializeFD) in order to use their FD
capabilities. In same way, the functions CAN_ReadFD (class-method: ReadFD) and CAN_WriteFD (class-method: WriteFD) have to be used for data transmission.

PCAN_BITRATE_ADAPTING

Access: RW

Description: This value is used to force an initialization process to succeed, even if the PCAN-Channel is being used by a PCAN-View with a different or unknown bit rate. The initialization function will return a PCAN_ERROR_CAUTION error, when the bit rate passed as parameter was different than that being used.

Possible values: The value must be one of the following values: PCAN_PARAMETER_OFF, PCAN_PARAMETER_ON. Note that other values will considered invalid.

Default value: Disabled (PCAN_PARAMETER_OFF).

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

REMARKS: This parameter can be set only on uninitialized channels. After connecting, an application can get the bit rate currently used by calling CAN_GetValue (class-method: GetValue) with the parameters PCAN_SPEED_QUERY, for standard CAN channels, or PCAN_SPEED_QUERY_FD for CAN-FD channels.

PCAN_BITRATE_INFO

Access: R

Description: This value is used to read the currently configured communication speed, as BTR0-BTR1 value, of a PCAN Channel connected as standard CAN.

Possible values: A number in the range [0..65535] (Word Value).

Default value: N/A.

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

**REMARKS**: This parameter can be used only on PCAN-Channels that have been initialized with the function [CAN_Initialize](#) (**class-method**: Initialize).

---

**PCAN_BITRATE_INFO_FD**

**Access**: R

**Description**: This value is used to read the currently configured communication speed, as a parameterized string value (FD bit rate string), of a PCAN Channel connected as CAN FD.

**Possible values**: a String representing a FD bit rate. See [FD Bit rate](#) for more information.

**Default value**: N/A.

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

**REMARKS**: This parameter can be used only on PCAN-Channels that have been initialized with the function [CAN_InitializeFD](#) (**class-method**: InitializeFD).

---

**PCAN_BUSSPEED_NOMINAL**

**Access**: R

**Description**: This value is used to read the currently configured nominal CAN Bus speed, as bits/second.

**Possible values**: a number representing the nominal CAN bus speed being used, as the amount of bits that can be transmitted in a second.

**Default value**: N/A.

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

**Access:**  

**Description:** This value is used to read the currently configured CAN data speed (Bit Rate Switch), as bits/second.

**Possible values:** a number representing the CAN data speed configured for BRS, as the amount of bits that can be transmitted in a second.

**Default value:** N/A.

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

**REMARKS:** The speed used for data transmission is the same as the nominal speed on devices that don’t support flexible data rate.

**PCAN_IP_ADDRESS**

**Access:**  

**Description:** This value is used to read the address used by a device for IP communication.

**Possible values:** a string representing the IP address of a device, in IPv4 format.

**Default value:** N/A.

**PCAN-Device:** PCAN-LAN (PCAN-Gateway Ethernet/Wireless devices).

**PCAN_LAN_SERVICE_STATUS**

**Access:**  

**Description:** This value is used to get the running status of the System-Service that is part of the Virtual PCAN-Gateway solution.

**Possible values:** This parameter can have one of these values: SERVICE_STATUS_RUNNING, SERVICE_STATUS_STOPPED.
**Default value:** N/A.

**PCAN-Device:** Default channel Only (PCAN_NONEBUS).

**REMARKS:** This parameter is only relevant in PCAN-LAN environments (using PCAN-Gateway Ethernet/Wireless devices).

---

**PCAN_ALLOW_STATUS_FRAMES**

**Access:** **RW**

**Description:** This parameter helps the user to allow / disallow the reception of messages of type "PCAN_MESSAGE_STATUS" within a PCAN Channel. When "PCAN_ALLOW_STATUS_FRAMES" is active (ON), generated Status messages are forwarded from the driver to the user application through the **CAN_Read/CAN_ReadFD** functions (**class-methods:** Read, ReadFD). Otherwise, the reception of Status frames is deactivated within the driver for this specific user application.

**Possible values:** The value must be one of the following values: **PCAN_PARAMETER_OFF**, **PCAN_PARAMETER_ON**. Note that other values will considered invalid.

**Default value:** Enabled (PCAN_PARAMETER_ON).

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

---

**PCAN_ALLOW_RTR_FRAMES**

**Access:** **RW**

**Description:** This parameter helps the user to allow / disallow the reception of messages of type "PCAN_MESSAGE_RTR" within a PCAN Channel. When "PCAN_ALLOW_RTR_FRAMES" is active (ON), incoming RTR messages are forwarded to the user application through the **CAN_Read/CAN_ReadFD** functions (**class-methods:** Read, ReadFD). Otherwise, the reception of RTR frames is deactivated within the driver for this specific user application.
Possible values: The value must be one of the following values:
PCAN_PARAMETER_OFF, PCAN_PARAMETER_ON. Note that other values will considered invalid.

Default value: Enabled (PCAN_PARAMETER_ON).

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

---

PCAN_ALLOW_ERROR_FRAMES

Access: R W

Description: This parameter helps the user to allow / disallow the reception of messages of type "PCAN_MESSAGE_ERRFRAME" within a PCAN Channel. When "PCAN_ALLOW_ERROR_FRAMES" is active (ON), generated Error messages are forwarded from the driver to the user application through the CAN_Read/CAN_ReadFD functions (class-methods: Read, ReadFD). Otherwise, the reception of Error frames is deactivated within the driver for this specific user application.

Possible values: The value must be one of the following values:
PCAN_PARAMETER_OFF, PCAN_PARAMETER_ON. Note that other values will considered invalid.

Default value: Disabled (PCAN_PARAMETER_OFF).

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

---

PCAN_INTERFRAME_DELAY

Access: R W

Description: This parameter is used to configure a delay, in microseconds, between sending frames. When this value is value is greater than 0, the driver includes that value as a pause between each written CAN frame. Otherwise, the CAN frames are sent as fast as possible.
**Possible values**: The parameter has a value range between [0..n], where n is the maximum value supported by the Firmware. If the maximum value supported by the firmware is lower than the entered one, the value will be truncated.

**Default value**: 0 (disabled).

**PCAN-Device**: All FPGA based PCAN devices.

---

**PCAN_ACCEPTANCE_FILTER_11BIT**

**Access**: R W

**Description**: This parameter is used to configure the reception filter of a PCAN channel with a specific 11-bit acceptance code and mask, as specified for the acceptance filter of the SJA1000 CAN controller. The acceptance code and mask are coded together in a 64-bit value, each of them using 4 bytes (Intel/Little-Endian format). The acceptance code is stored at the most significant bytes.

**Possible values**: Both parameter parts, code and mask, have a value range between [0..16838]. This means, the maximum value of this parameter as 64-bit value is 7036449226751, that is, hexadecimal 00003FFF00003FFFh. The mask uses the bit value '1' as "don't care bit".

**Default value**: 00000000000007FFh (no filtering).

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

**REMARKS**: This parameter is particularly adapted to the SJA1000 CAN controller. Reception filters can also be configured with the function `CAN_FilterMessages` (class-method: FilterMessages). Note that after a PCAN Channel is initialized, the status of its filter is fully opened. According with the current filter status, setting this parameter causes the following behavior:

- Filter status is PCAN_FILTER_OPEN: The filter is automatically closed and then configured with the given acceptance filter.
- Filter status is PCAN_FILTER_CLOSE: The filter is set to the
given acceptance filter.
- Filter status is PCAN_FILTER_CUSTOM: The filter is expanded with the given acceptance filter. If a different acceptance code is required instead of expanding the current one, the filter has to be closed first before setting the acceptance filter. To do this use the parameter PCAN_MESSAGE_FILTER.

PCAN_ACCEPTANCE_FILTER_29BIT

Access: **RW**

**Description:** This parameter is used to configure the reception filter of a PCAN channel with a specific 29-bit acceptance code and mask, as specified for the acceptance filter of the SJA1000 CAN controller. The acceptance code and mask are coded together in a 64-bit value, each of them using 4 bytes (Intel/Little-Endian format). The acceptance code is stored at the most significant bytes.

**Possible values:** Both parameter parts, code and mask, have a value range between [0..4294967295]. This means, the maximum value of this parameter as 64-bit value is 18446744073709551615, that is, hexadecimal FFFFFFFFFFFFFFFFFFFh. The mask uses the bit value '1' as "don't care bit".

**Default value:** 00000001FFFFFFFFh (no filtering).

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

**REMARKS:** This parameter is particularly adapted to the SJA1000 CAN controller. Reception filters can also be configured with the function CAN_FilterMessages(class-method: FilterMessages). Note that after a PCAN Channel is initialized, the status of its filter is fully opened. According to the current filter status, setting this parameter causes the following behavior:

- Filter status is PCAN_FILTER_OPEN: The filter is automatically closed and then configured with the given acceptance filter.
- Filter status is PCAN_FILTER_CLOSE: The filter is set to the given acceptance filter.
Filter status is PCAN_FILTER_CUSTOM: The filter is expanded with the given acceptance filter. If a different acceptance code is required instead of expanding the current one, the filter has to be closed first before setting the acceptance filter. To do this use the parameter PCAN_MESSAGE_FILTER.

See Also

CAN_GetValue (class-method: GetValue)
CAN_SetValue (class-method: SetValue)

Parameter Value Definitions

Naming Files, Paths, and Namespaces
TPCANMessageType

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

Syntax

**C++**

```cpp
#define TPCANMessageType BYTE
#define PCAN_MESSAGE_STANDARD 0x00
#define PCAN_MESSAGE_RTR 0x01
#define PCAN_MESSAGE_EXTENDED 0x02
#define PCAN_MESSAGE_FD 0x04
#define PCAN_MESSAGE_BRS 0x08
#define PCAN_MESSAGE_ESI 0x10
#define PCAN_MESSAGE_ERRFRAME 0x40
#define PCAN_MESSAGE_STATUS 0x80
```

**Pascal OO**

```pascal
{$Z1}
TPCANMessageType = (  
  PCAN_MESSAGE_STANDARD = $00,
  PCAN_MESSAGE_RTR = $01,
  PCAN_MESSAGE_EXTENDED = $02,
  PCAN_MESSAGE_FD = $04,
  PCAN_MESSAGE_BRS = $08,
  PCAN_MESSAGE_ESI = $10,
  PCAN_MESSAGE_ERRFRAME = $40,
  PCAN_MESSAGE_STATUS = $80
);
```

**C#**
[Flags]
public enum TPCANMessageType : byte
{
    PCAN_MESSAGE_STANDARD = 0x00,
    PCAN_MESSAGE_RTR = 0x01,
    PCAN_MESSAGE_EXTENDED = 0x02,
    PCAN_MESSAGE_FD = 0x04,
    PCAN_MESSAGE_BRS = 0x08,
    PCAN_MESSAGE_ESI = 0x10,
    PCAN_MESSAGE_ERRFRAME = 0x40,
    PCAN_MESSAGE_STATUS = 0x80,
}

[C++ / CLR]
[Flags]
public enum class TPCANMessageType : Byte
{
    PCAN_MESSAGE_STANDARD = 0x00,
    PCAN_MESSAGE_RTR = 0x01,
    PCAN_MESSAGE_EXTENDED = 0x02,
    PCAN_MESSAGE_FD = 0x04,
    PCAN_MESSAGE_BRS = 0x08,
    PCAN_MESSAGE_ESI = 0x10,
    PCAN_MESSAGE_ERRFRAME = 0x40,
    PCAN_MESSAGE_STATUS = 0x80,
};

[Visual Basic]
<Flags()>
Public Enum TPCANMessageType As Byte
    PCAN_MESSAGE_STANDARD = &H0
    PCAN_MESSAGE_RTR = &H1
    PCAN_MESSAGE_EXTENDED = &H2
    PCAN_MESSAGE_FD = &H4
PCAN_MESSAGE_BRS = &H8
PCAN_MESSAGE_ESI = &H10
PCAN_MESSAGE_ERRFRAME = &H40
PCAN_MESSAGE_STATUS = &H80

End Enum

Python

TPCANMessageType = c_ubyte

PCAN_MESSAGE_STANDARD = TPCANMessageType(0x00)
PCAN_MESSAGE_RTR = TPCANMessageType(0x01)
PCAN_MESSAGE_EXTENDED = TPCANMessageType(0x02)
PCAN_MESSAGE_FD = TPCANMessageType(0x04)
PCAN_MESSAGE_BRS = TPCANMessageType(0x08)
PCAN_MESSAGE_ESI = TPCANMessageType(0x10)
PCAN_MESSAGE_ERRFRAME = TPCANMessageType(0x40)
PCAN_MESSAGE_STATUS = TPCANMessageType(0x80)

Remarks

Several message types can be combined (Bit mask).

Note that messages with type PCAN_MESSAGE_FD, PCAN_MESSAGE_BRS, PCAN_MESSAGE_ESI, or a combination of them, can only be sent/received using the FD functions CAN_ReadFD and CAN_WriteFD (class-methods: ReadFD, WriteFD).

Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_MESSAGE_STANDARD</td>
<td>0</td>
<td>The PCAN message is a CAN Standard Frame (11-bit identifier).</td>
</tr>
<tr>
<td>PCAN_MESSAGE_RTR</td>
<td>1</td>
<td>The PCAN message</td>
</tr>
<tr>
<td>PCAN_MESSAGE_EXTENDED</td>
<td>2</td>
<td>The PCAN message is a CAN Extended Frame (29-bit identifier).</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>PCAN_MESSAGE_FD</td>
<td>4</td>
<td>The PCAN message represents a FD frame in terms of CiA specifications.</td>
</tr>
<tr>
<td>PCAN_MESSAGE_BRS</td>
<td>8</td>
<td>The PCAN message represents a FD bit rate switch (CAN data at a higher bit rate).</td>
</tr>
<tr>
<td>PCAN_MESSAGE_ESI</td>
<td>16</td>
<td>The PCAN message represents a FD error state indicator (CAN FD transmitter was error active).</td>
</tr>
<tr>
<td>PCAN_MESSAGE_ERRFRAME</td>
<td>64</td>
<td>The PCAN message represents an error frame. See <a href="#">Error Frames</a> for more information.</td>
</tr>
<tr>
<td>PCAN_MESSAGE_STATUS</td>
<td>128</td>
<td>The PCAN message represents a PCAN status message.</td>
</tr>
</tbody>
</table>

- **See Also**
  - [CAN_Read](#) (class-method: Read)
  - [CAN_ReadFD](#) (class-method: ReadFD)
CAN_Write (class-method: Write)
CAN_WriteFD (class-method: WriteFD)
TPCANType

Represents the type of PCAN (not plug&play) 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 TPCANType BYTE

#define PCAN_TYPE_ISA 0x01
#define PCAN_TYPE_ISA_SJA 0x09
#define PCAN_TYPE_ISA_PHYTEC 0x04
#define PCAN_TYPE_DNG 0x02
#define PCAN_TYPE_DNG_EPP 0x03
#define PCAN_TYPE_DNG_SJA 0x05
#define PCAN_TYPE_DNG_SJA_EPP 0x06
```

**Pascal OO**

```pascal
{$Z1}
TPCANType = (    PCAN_TYPE_ISA = $01,
    PCAN_TYPE_ISA_SJA = $09,
    PCAN_TYPE_ISA_PHYTEC = $04,
    PCAN_TYPE_DNG = $02,
    PCAN_TYPE_DNG_EPP = $03,
    PCAN_TYPE_DNG_SJA = $05,
    PCAN_TYPE_DNG_SJA_EPP = $06
);
```

**C#**

```csharp
public enum TPCANType : byte
```
```{json}
{
    PCAN_TYPE_Isa = 0x01,
    PCAN_TYPE_Isa_Sja = 0x09,
    PCAN_TYPE_Isa_Phytec = 0x04,
    PCAN_TYPE_Dng = 0x02,
    PCAN_TYPE_Dng_Epp = 0x03,
    PCAN_TYPE_Dng_Sja = 0x05,
    PCAN_TYPE_Dng_Sja_Epp = 0x06,
}
```

```csharp
public enum class TPCANType : Byte
{
    PCAN_TYPE_Isa = 0x01,
    PCAN_TYPE_Isa_Sja = 0x09,
    PCAN_TYPE_Isa_Phytec = 0x04,
    PCAN_TYPE_Dng = 0x02,
    PCAN_TYPE_Dng_Epp = 0x03,
    PCAN_TYPE_Dng_Sja = 0x05,
    PCAN_TYPE_Dng_Sja_Epp = 0x06,
};
```

```vbnet
Public Enum TPCANType As Byte
    PCAN_TYPE_Isa = &H1
    PCAN_TYPE_Isa_Sja = &H9
    PCAN_TYPE_Isa_Phytec = &H4
    PCAN_TYPE_Dng = &H2
    PCAN_TYPE_Dng_Epp = &H3
    PCAN_TYPE_Dng_Sja = &H5
    PCAN_TYPE_Dng_Sja_Epp = &H6
End Enum
```

```python
```
TPCANType = c_ubyte

PCAN_TYPE_ISA = TPCANType(0x01)
PCAN_TYPE_ISA_SJA = TPCANType(0x09)
PCAN_TYPE_ISA_PHYTEC = TPCANType(0x04)
PCAN_TYPE_DNG = TPCANType(0x02)
PCAN_TYPE_DNG_EPP = TPCANType(0x03)
PCAN_TYPE_DNG_SJA = TPCANType(0x05)
PCAN_TYPE_DNG_SJA_EPP = TPCANType(0x06)

### Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_TYPE_ISA</td>
<td>1</td>
<td>PCAN-ISA 82C200.</td>
</tr>
<tr>
<td>PCAN_TYPE_ISA_SJA</td>
<td>9</td>
<td>PCAN-ISA SJA1000.</td>
</tr>
<tr>
<td>PCAN_TYPE_ISA_PHYTEC</td>
<td>4</td>
<td>PHYTEC ISA.</td>
</tr>
<tr>
<td>PCAN_TYPE_DNG</td>
<td>2</td>
<td>PCAN-Dongle 82C200.</td>
</tr>
<tr>
<td>PCAN_TYPE_DNG_EPP</td>
<td>3</td>
<td>PCAN-Dongle EPP 82C200.</td>
</tr>
<tr>
<td>PCAN_TYPE_DNG_SJA</td>
<td>5</td>
<td>PCAN-Dongle SJA1000.</td>
</tr>
<tr>
<td>PCAN_TYPE_DNG_SJA_EPP</td>
<td>6</td>
<td>PCAN-Dongle EPP SJA1000.</td>
</tr>
</tbody>
</table>

### See Also

PCAN_Initialize (class-method: Initialize)
TPCANMode

Represents a PCAN filter mode. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

### C++

```cpp
#define TPCANMode BYTE
#define PCAN_MODE_STANDARD PCAN_MESSAGE_STANDARD
#define PCAN_MODE_EXTENDED PCAN_MESSAGE_EXTENDED
```

### Pascal OO

```pascal
{$Z1}
TPCANMode = (
    PCAN_MODE_STANDARD = Byte(PCAN_MESSAGE_STANDARD),
    PCAN_MODE_EXTENDED = Byte(PCAN_MESSAGE_EXTENDED);
```

### C#

```csharp
public enum TPCANMode : byte
{
    PCAN_MODE_STANDARD = TPCANMessageType.PCAN_MESSAGE_STANDARD,
    PCAN_MODE_EXTENDED = TPCANMessageType.PCAN_MESSAGE_EXTENDED
}
```

### C++ / CLR

```csharp
public enum class TPCANMode : Byte
{
    PCAN_MODE_STANDARD = TPCANMessageType::PCAN_MESSAGE_STANDARD,
    PCAN_MODE_EXTENDED = TPCANMessageType::PCAN_MESSAGE_EXTENDED
};
```
Visual Basic

Public Enum TPCANMode As Byte
    PCAN_MODE_STANDARD = TPCANMessageType.PCAN_MESSAGE_STANDARD
    PCAN_MODE_EXTENDED = TPCANMessageType.PCAN_MESSAGE_EXTENDED
End Enum

Python

TPCANMode = c_ubyte

PCAN_MODE_STANDARD = PCAN_MESSAGE_STANDARD
PCAN_MODE_EXTENDED = PCAN_MESSAGE_EXTENDED

Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_MODE_STANDARD</td>
<td>0</td>
<td>Mode is Standard (11-bit identifier).</td>
</tr>
<tr>
<td>PCAN_MODE_EXTENDED</td>
<td>2</td>
<td>Mode is Extended (29-bit identifier).</td>
</tr>
</tbody>
</table>

See Also

CAN_FilterMessages (class-method: FilterMessages)
TPCANBaudrate

Represents a PCAN bit rate register value. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

### C++

```c++
#define TPCANBaudrate WORD
#define PCAN_BAUD_1M 0x0014
#define PCAN_BAUD_800K 0x0016
#define PCAN_BAUD_500K 0x001C
#define PCAN_BAUD_250K 0x011C
#define PCAN_BAUD_125K 0x031C
#define PCAN_BAUD_100K 0x432F
#define PCAN_BAUD_95K 0xC34E
#define PCAN_BAUD_83K 0x852B
#define PCAN_BAUD_50K 0x472F
#define PCAN_BAUD_47K 0x1414
#define PCAN_BAUD_33K 0x8B2F
#define PCAN_BAUD_20K 0x532F
#define PCAN_BAUD_10K 0x672F
#define PCAN_BAUD_5K 0x7F7F
```

### Pascal OO

```pascal
{$Z2}
TPCANBaudrate = ( 
    PCAN_BAUD_1M = $0014,
    PCAN_BAUD_800K = $0016,
    PCAN_BAUD_500K = $001C,
    PCAN_BAUD_250K = $011C,
```
PCAN_BAUD_125K = $031C,
PCAN_BAUD_100K = $432F,
PCAN_BAUD_95K = $C34E,
PCAN_BAUD_83K = $852B,
PCAN_BAUD_50K = $472F,
PCAN_BAUD_47K = $1414,
PCAN_BAUD_33K = $8B2F,
PCAN_BAUD_20K = $532F,
PCAN_BAUD_10K = $672F,
PCAN_BAUD_5K = $7F7F

C#

```csharp
public enum TPCANBaudrate : ushort {
    PCAN_BAUD_1M = 0x0014,
    PCAN_BAUD_800K = 0x0016,
    PCAN_BAUD_500K = 0x001C,
    PCAN_BAUD_250K = 0x011C,
    PCAN_BAUD_125K = 0x031C,
    PCAN_BAUD_100K = 0x432F,
    PCAN_BAUD_95K = 0xC34E,
    PCAN_BAUD_83K = 0x852B,
    PCAN_BAUD_50K = 0x472F,
    PCAN_BAUD_47K = 0x1414,
    PCAN_BAUD_33K = 0x8B2F,
    PCAN_BAUD_20K = 0x532F,
    PCAN_BAUD_10K = 0x672F,
    PCAN_BAUD_5K = 0x7F7F,
}
```

C++ / CLR

```c++
public enum class TPCANBaudrate : UInt16 {
}
```
PCAN_BAUD_1M = 0x0014,
PCAN_BAUD_800K = 0x0016,
PCAN_BAUD_500K = 0x001C,
PCAN_BAUD_250K = 0x011C,
PCAN_BAUD_125K = 0x031C,
PCAN_BAUD_100K = 0x432F,
PCAN_BAUD_95K = 0xC34E,
PCAN_BAUD_83K = 0x852B,
PCAN_BAUD_50K = 0x472F,
PCAN_BAUD_47K = 0x1414,
PCAN_BAUD_33K = 0x8B2F,
PCAN_BAUD_20K = 0x532F,
PCAN_BAUD_10K = 0x672F,
PCAN_BAUD_5K = 0x7F7F,

};

Visual Basic

Public Enum TPCANBaudrate As UInt16
    PCAN_BAUD_1M = &H14
    PCAN_BAUD_800K = &H16
    PCAN_BAUD_500K = &H1C
    PCAN_BAUD_250K = &H11C
    PCAN_BAUD_125K = &H31C
    PCAN_BAUD_100K = &H432F
    PCAN_BAUD_95K = &C34E
    PCAN_BAUD_83K = &852B
    PCAN_BAUD_50K = &H472F
    PCAN_BAUD_47K = &1414
    PCAN_BAUD_33K = &8B2F
    PCAN_BAUD_20K = &H532F
    PCAN_BAUD_10K = &H672F
    PCAN_BAUD_5K = &H7F7F
End Enum

Python
TPCANBaudrate = c_ushort

PCAN_BAUD_1M  =  TPCANBaudrate(0x0014)
PCAN_BAUD_800K =  TPCANBaudrate(0x0016)
PCAN_BAUD_500K =  TPCANBaudrate(0x001C)
PCAN_BAUD_250K =  TPCANBaudrate(0x011C)
PCAN_BAUD_125K =  TPCANBaudrate(0x031C)
PCAN_BAUD_100K =  TPCANBaudrate(0x432F)
PCAN_BAUD_95K  =  TPCANBaudrate(0xC34E)
PCAN_BAUD_83K  =  TPCANBaudrate(0x852B)
PCAN_BAUD_50K  =  TPCANBaudrate(0x472F)
PCAN_BAUD_47K  =  TPCANBaudrate(0x1414)
PCAN_BAUD_33K  =  TPCANBaudrate(0x8B2F)
PCAN_BAUD_20K  =  TPCANBaudrate(0x532F)
PCAN_BAUD_10K  =  TPCANBaudrate(0x672F)
PCAN_BAUD_5K   =  TPCANBaudrate(0x7F7F)

### Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_BAUD_1M</td>
<td>20</td>
<td>1 MBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_800K</td>
<td>22</td>
<td>800 kBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_500K</td>
<td>28</td>
<td>500 kBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_250K</td>
<td>284</td>
<td>250 kBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_125K</td>
<td>796</td>
<td>125 kBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_100K</td>
<td>17199</td>
<td>100 kBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_95K</td>
<td>49998</td>
<td>95,238 kBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_83K</td>
<td>34091</td>
<td>83,333 kBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_50K</td>
<td>18223</td>
<td>50 kBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_47K</td>
<td>5140</td>
<td>47,619 kBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_33K</td>
<td>35631</td>
<td>33,333 kBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_20K</td>
<td>21295</td>
<td>20 kBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_10K</td>
<td>26415</td>
<td>10 kBit/s.</td>
</tr>
<tr>
<td>PCAN_BAUD_5K</td>
<td>32639</td>
<td>5 kBit/s.</td>
</tr>
</tbody>
</table>

### See Also

[CAN_Initialize](#) (class-method: [Initialize](#))
TPCANBitrateFD

Represents a bit rate string with flexible data rate (FD).

Syntax

C++
#define TPCANBitrateFD LPSTR

Pascal OO
TPCANBitrateFD = String;

C#
using TPCANBitrateFD = System.String;

C++ / CLR
#define TPCANBitrateFD System::String^

Visual Basic
Imports TPCANBitrateFD = System.String

Python
TPCANBitrateFD = c_char_p

Remarks

.NET Framework programming languages:

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

Aliases are defined in the Peak.Can.Basic 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.Basic Namespace. Otherwise, just use the native type, which in this case is a String.

C#:

```csharp
using System;
using Peak.Can.Basic;
using TPCANBitrateFD = System.String; // Alias's declaration for System.String
```

Visual Basic:

```vbnet
Imports System
Imports Peak.Can.Basic
Imports TPCANBitrateFD = System.String ' Alias declaration for System.String
```

See Also

FB Bit rate Parameter Definitions
TPCANTimestampFD

Represents the timestamp of a CAN message with flexible data rate. The time-stamp contains the number of microseconds since the start of Windows.

Syntax

- **C++**
  ```
  #define TPCANTimestampFD UINT64
  ```

- **Pascal OO**
  ```
  TPCANTimestampFD = UInt64;
  ```

- **C#**
  ```
  using TPCANTimestampFD = System.UInt64;
  ```

- **C++ / CLR**
  ```
  #define TPCANTimestampFD System::UInt64
  ```

- **Visual Basic**
  ```
  Imports TPCANTimestampFD = System.UInt64
  ```

- **Python**
  ```
  TPCANTimestampFD = c_ulonglong
  ```

Remarks

**.NET Framework programming languages:**

An alias is used to represent a timestamp for flexible data rate messages under Microsoft .NET in order to originate an homogeneity between all programming languages listed above.

Aliases are defined in the [Peak.Can.Basic](#) 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.Basic Namespace. Otherwise, just use the native type, which in this case is a UInt64.

C#:

```csharp
using System;
using Peak.Can.Basic;
using TPCANTimestampFD = System.UInt64; // Alias's declaration for System.UInt64
```

Visual Basic:

```vb
Imports System
Imports Peak.Can.Basic
Imports TPCANTimestampFD = System.UInt64 ' Alias's declaration for System.UInt64
```
The methods defined for the classes **PCANBasic** and **TPCANBasic** are divided in 4 groups of functionality. Note that, with exception of the method version for Python, 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><strong>Initialize</strong></td>
<td>Initializes a PCAN Channel.</td>
</tr>
<tr>
<td><strong>InitializeFD</strong></td>
<td>Initializes a <strong>FD capable</strong> PCAN Channel.</td>
</tr>
<tr>
<td><strong>Uninitialize</strong></td>
<td>Uninitializes a PCAN Channel.</td>
</tr>
</tbody>
</table>

## Configuration

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SetValue</strong></td>
<td>Sets a configuration or information value within a PCAN Channel.</td>
</tr>
<tr>
<td><strong>FilterMessages</strong></td>
<td>Configures the message's reception filter of a PCAN Channel.</td>
</tr>
</tbody>
</table>

## Information

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetValue</strong></td>
<td>Retrieves information from a PCAN</td>
</tr>
</tbody>
</table>
### Communication

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetStatus</strong></td>
<td>Retrieves the current BUS status of a PCAN Channel.</td>
</tr>
<tr>
<td><strong>GetErrorText</strong></td>
<td>Gets a descriptive text for an error code.</td>
</tr>
<tr>
<td><strong>Read</strong></td>
<td>Reads a CAN message from the receive queue of a PCAN Channel.</td>
</tr>
<tr>
<td><strong>ReadFD</strong></td>
<td>Reads a CAN message from the receive queue of a <strong>FD capable</strong> PCAN Channel.</td>
</tr>
<tr>
<td><strong>Write</strong></td>
<td>Transmits a CAN message using a connected PCAN Channel.</td>
</tr>
<tr>
<td><strong>WriteFD</strong></td>
<td>Transmits a CAN message using a connected <strong>FD capable</strong> PCAN Channel.</td>
</tr>
<tr>
<td><strong>Reset</strong></td>
<td>Resets the receive and transmit queues of a PCAN Channel.</td>
</tr>
</tbody>
</table>
Initialize

Initializes a PCAN Channel.

---

**Overloads**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize(TPCANHandle, TPCANBaudrate)</td>
<td>Initializes a Plug-And-Play PCAN Channel.</td>
</tr>
<tr>
<td>Initialize(TPCANHandle, TPCANBaudrate, TPCANType, UInt32, UInt16)</td>
<td>Initializes a Not-Plug-And-Play PCAN Channel.</td>
</tr>
</tbody>
</table>
**Initialize(TPCANHandle, TPCANBaudrate)**

Initializes a PCAN Channel which represents a Plug & Play PCAN-Device.

**Syntax**

**Pascal OO**

```pascal
class function Initialize(
    Channel: TPCANHandle;
    Btr0Btr1: TPCANBaudrate
): TPCANStatus; overload;
```

**C#**

```csharp
public static extern TPCANStatus Initialize(
    TPCANHandle Channel,
    TPCANBaudrate Btr0Btr1);
```

**C++/CLR**

```cpp
static TPCANStatus Initialize(
    TPCANHandle Channel,
    TPCANBaudrate Btr0Btr1);
```

**Visual Basic**

```vbnet
Public Shared Function Initialize( _
    ByVal Channel As TPCANHandle, _
    ByVal Btr0Btr1 As TPCANBaudrate) As TPCANStatus
End Function
```

**Python**

```python
def Initialize(  
    self,  
    Channel,  
    Btr0Btr1)
```
### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>Btr0Btr1</td>
<td>The speed for the communication (BTR0BTR1 code).</td>
</tr>
</tbody>
</table>

### Returns

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

<table>
<thead>
<tr>
<th>PCAN_ERROR_CAUTION:</th>
<th>Indicates that the channel has been initialized but at a different bit rate as the given one.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_ILLHANDLE:</td>
<td>Indicates that the desired PCAN Channel is not valid. Check the list of valid Channels.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLHW:</td>
<td>Indicates that the desired PCAN Channel is not available.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLOPERATION:</td>
<td>Indicates that an action cannot be executed due to the state of the hardware. Possible causes are:</td>
</tr>
<tr>
<td></td>
<td>• The desired PCAN-Channel is a LAN Channel, which uses a</td>
</tr>
</tbody>
</table>
different bit rate than the specified.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the desired PCAN Channel cannot be connected because it is already in use (PCAN-Basic / PCAN-Light environment).</td>
</tr>
<tr>
<td>PCAN_ERROR_NETINUSE:</td>
<td>Indicates that the desired PCAN-Channel is being used with a different bit rate (PCAN-View).</td>
</tr>
<tr>
<td>PCAN_ERROR_HWINUSE:</td>
<td>Indicates that the desired PCAN-Channel is being used (CanApi2 connection).</td>
</tr>
<tr>
<td>PCAN_ERROR_NODRIVER:</td>
<td>The driver needed for connecting the desired PCAN Channel is not loaded.</td>
</tr>
</tbody>
</table>

Remarks

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

Initializing a PCAN Channel means:

- to reserve the Channel for the calling application/process.
- to allocate channel resources, like receive and transmit queues.
- to register/connect the Hardware denoted by the channel handle.
- **to check and adapt the bus speed, if the Channel is already**
in use. *(Only if the Channel was pre-configured as Bitrate Adapting; see: Bitrate-Adapting Parameter).*

- **to set the channel in Listen-Only mode.** *(Only if the channel was pre-configured as Listen-Only; see: Listen-Only Parameter).*
- to open the messages filter for the application/process.
- to set-up the default values of the different parameters (See GetValue).
- **to set the Receive Status of the channel.** *(Pre-configured value; see: Receive Status Parameter).*

Different than the PCAN-Light API, the Initialization process will fail if an application try to initialize a PCAN-Channel that has been initialized already within the same process.

Take in consideration that initializing a channel causes a reset of the CAN hardware, when the bus status is other than OK. In this way errors like BUSOFF, BUSHEAVY, and BUSLIGHT, are removed.

**PCAN-LAN Channels**

A PCAN-LAN channel doesn’t allow changing the bit rate using PCAN-Basic. In order to connect a PCAN-LAN Channel it is necessary to know the bit rate of the PCAN-Gateway device that is represented by that channel. If the bit rate is not known, the parameter **Bitrate-Adapting** should be used.

**Python Notes**

- **Class-Method:** Different than the .NET Framework, under Python a variable has to be instantiated with an object of type **PCANBasic** in order to use the API functionality.
- **Python's first argument convention:** Under Python, ‘self’ is a parameter that is automatically included within the call of this method, within a **PCANBasic** object and hasn't to be indicated in a method call. This parameter represents the calling object itself.
- **Plug-&-Play and No-Plug-&-Play hardware:** In order to initialize a channel which represents a Plug-&-Play PCAN device, only the Channel-handle and bit rate parameters are needed. The other parameters will be assigned their default values. For No-Plug-&-Play devices, all parameters have to be entered.
Example

The following example shows the initialize and uninitialize processes for a Plug-And-Play channel (channel 2 of a PCAN-PCI hardware). In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

C#:

```csharp
TPCANStatus result;
StringBuilder strMsg;

// The Plug & Play Channel (PCAN-PCI) is initialized
result = PCANBasic.Initialize(PCANBasic.PCAN_PCIBUS2)
if (result != TPCANStatus.PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    strMsg = new StringBuilder(256);
    PCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox.Show(strMsg.ToString());
}
else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized

// All initialized channels are released
//
PCANBasic.Uninitialize(PCANBasic.PCAN_NONEBUS);
```

C++/CLR:

```csharp
TPCANStatus result;
StringBuilder^ strMsg;

// The Plug & Play Channel (PCAN-PCI) is initialized
result = PCANBasic::Initialize(PCANBasic::PCAN_PCI
```
if (result != TPCANStatus::PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    strMsg = gcnew StringBuilder(256);
    PCANBasic::GetErrorText(result, 0, strMsg);
    MessageBox::Show(strMsg->ToString());
}
else
    MessageBox::Show("PCAN-PCI (Ch-2) was initialized.

// All initialized channels are released
//
PCANBasic::Uninitialize(PCANBasic::PCAN_NONEBUS);

Visual Basic:

Dim result As TPCANStatus
Dim strMsg As StringBuilder

' The Plug & Play Channel (PCAN-PCI) is initialized
result = PCANBasic. Initialize(PCANBasic.PCAN_PCIBUS)
If result <> TPCANStatus.PCAN_ERROR_OK Then
' An error occurred, get a text describing the error
    strMsg = New StringBuilder(256)
    PCANBasic.GetErrorText(result, 0, strMsg)
    MessageBox.Show(strMsg.ToString)
Else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized.
End If

' All initialized channels are released
PCANBasic.Uninitialize(PCANBasic.PCAN_NONEBUS)

Pascal OO:
```
var
result : TPCANStatus;
strMsg: array [0..256] of Char;
begin
  // The Plug & Play Channel (PCAN-PCI) is initialized
  result := TPCANBasic.Initialize(TPCANBasic.PCAN_PCIBUS2)
  if (result <> PCAN_ERROR_OK) then
  begin
    // An error occurred, get a text describing the error and show it
    TPCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox(0, strMsg, 'Error', MB_OK);
  end
  else
  begin
    MessageBox(0, 'PCAN-PCI (Ch-2) was initialized'
    // All initialized channels are released
    TPCANBasic.Uninitialize(TPCANBasic.PCAN_NONEBUS)
  end;
end;
```

Python:

```
# The Plug & Play Channel (PCAN-PCI) is initialized
#
objPCAN = PCANBasic()
result = objPCAN.Initialize(PCAN_PCIEBUS2, PCAN_BAUD_500K)
if result != PCAN_ERROR_OK:
  # An error occurred, get a text describing the error
  result = objPCAN.GetErrorText(result)
  print result[1]
else:
  print "PCAN-PCI (Ch-2) was initialized"

# All initialized channels are released
```
objPCAN.Uninitialize(PCAN_NONEBUS)

See Also

Uninitialize
GetValue
Understanding PCAN-Basic

Plain function Version: CAN_Initialize
Initialize(TPCANHandle, TPCANBaudrate, TPCANType, UInt32, UInt16)

Initializes a PCAN Channel which represents a Not Plug & Play PCAN-Device.

Syntax

**Pascal OO**

```pascal
class function Initialize(
    Channel: TPCANHandle;
    Btr0Btr1: TPCANBaudrate;
    HwType: TPCANType;
    IOPort: LongWord;
    Interrupt: Word
): TPCANStatus; overload;
```

**C#**

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_Initialize")
public static extern TPCANStatus Initialize(
    [MarshalAs(UnmanagedType.U1)]
    TPCANHandle Channel,
    [MarshalAs(UnmanagedType.U2)]
    TPCANBaudrate Btr0Btr1,
    [MarshalAs(UnmanagedType.UnmanagedType)]
    TPCANType HwType,
    UInt32 IOPort,
    UInt16 Interrupt);
```

**C++ / CLR**

```csharp
[DllImport("PCANBasic.dll",EntryPoint = "CAN_")
static TPCANStatus Initialize(
    [MarshalAs(UnmanagedType::U1)]
    TPCANHandle Channel,
    [MarshalAs(UnmanagedType::U2)]
    TPCANBaudrate Btr0Btr1,
    [MarshalAs(UnmanagedType::UnmanagedType)]
    TPCANType HwType,
    UInt32 IOPort,
    UInt16 Interrupt);
```
**Visual Basic**

```vbnet
<DllImport("PCANBasic.dll", EntryPoint:="CAN_Initiate")
Public Shared Function Initialize(
    ByVal Channel As TPCANHandle,
    ByVal Btr0Btr1 As TPCANBaudrate,
    ByVal HwType As TPCANType,
    ByVal IOPort As UInt32,
    ByVal Interrupt As UInt16) As TPCANStatus
End Function
```

**Python**

```python
def Initialize(
    self,
    Channel,
    Btr0Btr1,
    HwType = TPCANType(0),
    IOPort = c_uint(0),
    Interrupt = c_ushort(0))
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel</td>
</tr>
<tr>
<td><strong>Btr0Btr1</strong></td>
<td>The speed for the communication (BTR0BTR1 code).</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><strong>HwType</strong></td>
<td>The type of hardware and operation mode (see TPCANMode).</td>
</tr>
<tr>
<td><strong>IOPort</strong></td>
<td>The I/O address for the parallel port.</td>
</tr>
<tr>
<td><strong>Interrupt</strong></td>
<td>Interrupt number of the parallel port.</td>
</tr>
</tbody>
</table>

### Returns

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

<table>
<thead>
<tr>
<th><strong>PCAN_ERROR_CAUTION:</strong></th>
<th>Indicates that the channel has been initialized but at a different bit rate as the given one.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PCAN_ERROR_ILLHANDLE:</strong></td>
<td>Indicates that the desired PCAN Channel is not valid. Check the list of valid Channels.</td>
</tr>
<tr>
<td><strong>PCAN_ERROR_ILLHW:</strong></td>
<td>Indicates that the desired PCAN Channel is not available.</td>
</tr>
<tr>
<td><strong>PCAN_ERROR_ILLOperation:</strong></td>
<td>Indicates that an action cannot be executed due to the state of the hardware. Possible causes are:</td>
</tr>
</tbody>
</table>
- The desired PCAN-Channel is a LAN Channel, which uses a different bit rate than the specified.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_REGTEST:</td>
<td>Indicates a problem with hardware registration, normally due to wrong values in the parameters 'HwType', 'IOPort' and 'Interrupt'.</td>
</tr>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the desired PCAN channel cannot be connected because it is already in use (PCAN-Basic / PCAN-Light environment).</td>
</tr>
<tr>
<td>PCAN_ERROR_NETINUSE:</td>
<td>Indicates that the desired PCAN-Channel is being used with a different bit rate (PCAN-View).</td>
</tr>
<tr>
<td>PCAN_ERROR_HWINUSE:</td>
<td>Indicates that the desired PCAN-Channel is being used (CanApi2 connection).</td>
</tr>
<tr>
<td>PCAN_ERROR_NODRIVER:</td>
<td>The driver needed for connecting the desired PCAN Channel is not loaded.</td>
</tr>
</tbody>
</table>

Remarks

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

Initializing a PCAN Channel means:

- to reserve the Channel for the calling application/process.
- to allocate channel resources, like receive and transmit queues.
- to register/connect the Hardware denoted by the channel handle.
- **to check and adapt the bus speed, if the Channel is already in use.** *(Only if the Channel was pre-configured as Bitrate Adapting; see: Bitrate-Adapting Parameter).*
- **to set the channel in Listen-Only mode.** *(Only if the channel was pre-configured as Listen-Only; see: Listen-Only Parameter).*
- to configure the filter to catch all messages being transmitted in the bus.
- to set-up the default values of the different parameters *(See GetValue).*
- **to set the Receive Status of the channel.** *(Pre-configured value; see: Receive Status Parameter).*

Different than the PCAN-Light API, the Initialization process will fail if an application try to initialize a PCAN-Channel that has been initialized already within the same process.

Take in consideration that initializing a channel causes a reset of the CAN hardware, when the bus status is other than OK. In this way errors like BUSOFF, BUSWARNING, and BUSPASSIVE, are removed.

**PCAN-LAN Channels**

A PCAN-LAN channel doesn't allow changing the bit rate using PCAN-Basic. In order to connect a PCAN-LAN Channel it is necessary to know the bit rate of the PCAN-Gateway device that is represented by that channel. If the bit rate is not known, the parameter Bitrate-Adapting should be used.

**Python Notes**

- Class-Method: Unlike the .NET Framework, under Python a variable has to be instantiated with an object of type PCANBasic in order to use the API functionality.
• Python's first argument convention: Under Python, 'self' is a parameter that is automatically included within the call of this method, within a PCANBasic object and hasn't to be indicated in a method call. This parameter represents the calling object itself.
• Plug-&-Play and No-Plug-&-Play hardware: In order to initialize a channel which represents a Plug-&-Play PCAN device, only the Channel-handle and bit rate parameters are needed. The other parameters will be assigned their default values. For No-Plug-&-Play devices, all parameters have to be entered.

Example

The following example shows the initialize and uninitialize processes for a Not-Plug-And-Play channel (channel 1 of the PCAN-DNG). In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

C#:

```csharp
TPCANStatus result;
StringBuilder strMsg;

// The Not Plug & Play Channel (PCAN-DNG) is initia
result = PCANBasic.Initialize(PCANBasic.PCAN_DNGBUS1)
if (result != TPCANStatus.PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    strMsg = new StringBuilder(256);
    PCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox.Show(strMsg.ToString());
}
else
    MessageBox.Show("PCAN-DNG (Ch-1) was initialized"

// All initialized channels are released
```
PCANBasic.Uninitialize(PCANBasic.PCAN_NONEBUS);

C++/CLR:

```c++
TPCANStatus result;
StringBuilder^ strMsg;

// The Not Plug & Play Channel (PCAN-DNG) is initialized
result = PCANBasic::Initialize(PCANBasic::PCAN_DNGBUS1)
if (result != TPCANStatus::PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    strMsg = gcnew StringBuilder(256);
    PCANBasic::GetErrorText(result, 0, strMsg);
    MessageBox::Show(strMsg->ToString());
}
else
    MessageBox::Show("PCAN-DNG (Ch-1) was initialized")

// All initialized channels are released
PCANBasic::Uninitialize(PCANBasic::PCAN_NONEBUS);
```

Visual Basic:

```vb
Dim result As TPCANStatus
Dim strMsg As StringBuilder

' The Not Plug & Play Channel (PCAN-DNG) is initialized
result = PCANBasic.Initialize(PCANBasic.PCAN_DNGBUS1)
If result <> TPCANStatus.PCAN_ERROR_OK Then
    ' An error occurred, get a text describing the error
    strMsg = New StringBuilder(256)
```
PCANBasic.GetErrorText(result, 0, strMsg)
MessageBox.Show(strMsg.ToString)
Else
    MessageBox.Show("PCAN-DNG (Ch-1) was initialized")
End If
' All initialized channels are released '
PCANBasic.Uninitialize(PCANBasic.PCAN_NONEBUS)

Pascal OO:

var
    result : TPCANStatus;
    strMsg: array [0..256] of Char;
begin
    // The Not Plug & Play Channel (PCAN-DNG) is initialized
    result := TPCANBasic.Initialize(TPCANBasic.PCAN_DNGBUS1);
    If (result <> PCAN_ERROR_OK) Then
        begin
            // An error occurred, get a text describing the error and show it
            TPCANBasic.GetErrorText(result, 0, strMsg);
            MessageBox(0, strMsg, 'Error', MB_OK);
        end
    else
        MessageBox(0, 'PCAN-DNG (Ch-1) was initialized'

        // All initialized channels are released
        //
        TPCANBasic.Uninitialize(TPCANBasic.PCAN_NONEBUS);
end;

Python:

# The Not Plug & Play Channel (PCAN-DNG) is initialized
#
objPCAN = PCANBasic()
result = objPCAN.Initialize(PCAN_DNGBUS1, PCAN_BAUD_500K)
if result != PCAN_ERROR_OK:
    # An error occurred, get a text describing the
    #
    result = objPCAN.GetErrorText(result)
    print result[1]
else:
    print "PCAN-DNG (Ch-1) was initialized"

# All initialized channels are released
#
objPCAN.Uninitialize(PCAN_NONEBUS)

See Also

Uninitialize
GetValue
Understanding PCAN-Basic

Plain function Version: CAN_Initialize
InitializeFD

Initializes a FD capable PCAN Channel.

Syntax

- **Pascal OO**

```pascal
class function InitializeFD(
    Channel: TPCANHandle;
    BitrateFD: TPCANBitrateFD
): TPCANStatus;
```

- **C#**

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_InitializeFD")
public static extern TPCANStatus InitializeFD(
    [MarshalAs(UnmanagedType.U1)]
    TPCANHandle Channel,
    TPCANBitrateFD BitrateFD);
```

- **C++ / CLR**

```cpp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_InitializeFD")
static TPCANStatus InitializeFD(
    [MarshalAs(UnmanagedType::U1)]
    TPCANHandle Channel,
    TPCANBitrateFD BitrateFD);
```

- **Visual Basic**

```vb
<DllImport("PCANBasic.dll", EntryPoint:="CAN_InitializeFD")>
Public Shared Function InitializeFD(
    <MarshalAs(UnmanagedType.U1)> _
    ByVal Channel As TPCANHandle, _
    ByVal BitrateFD As TPCANBitrateFD) As TPCANStatus
End Function
```
```python
def InitializeFD(self, Channel, BitrateFD)
```

### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a FD capable PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>BitrateFD</td>
<td>The speed for the communication (FD Bitrate string).</td>
</tr>
</tbody>
</table>

### Returns

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

- **PCAN_ERROR_CAUTION:** Indicates that the channel has been initialized but at a different bit rate as the given one.
- **PCAN_ERROR_ILLHANDLE:** Indicates that the desired PCAN Channel is not valid. Check the list of valid Channels.
- **PCAN_ERROR_ILLHW:** Indicates that the desired PCAN Channel is not available.
- **PCAN_ERROR_ILLOPERATION:** Indicates that an action
cannot be executed due to the state of the hardware. Possible causes are:

- The desired PCAN Channel is not FD capable and cannot be initialized using this method.
- The desired PCAN-Channel is a LAN Channel, which uses a different bit rate than the specified.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_INITIALIZE</td>
<td>Indicates that the desired PCAN Channel cannot be connected because it is already in use (PCAN-Basic / PCAN-Light environment).</td>
</tr>
<tr>
<td>PCAN_ERROR_NETINUSE</td>
<td>Indicates that the desired PCAN-Channel is being used with a different bit rate (PCAN-View).</td>
</tr>
<tr>
<td>PCAN_ERROR_HWINUSE</td>
<td>Indicates that the desired PCAN-Channel is being used (CanApi connection).</td>
</tr>
<tr>
<td>PCAN_ERROR_NODRIVER</td>
<td>The driver needed for connecting the desired PCAN Channel is not loaded.</td>
</tr>
</tbody>
</table>

Remarks

**Note on correspondence of methods:**
A Channel that is initialized using InitializeFD must use **ReadFD** and **WriteFD** for communication. Calling **Read** and/or **Write** will result in a **PCAN_ERROR_ILLOPERATION** error.

As indicated by its name, the InitializeFD method initiates a FD capable PCAN Channel, preparing it for communicate within the CAN bus connected to it. Calls to the API methods will fail if they are used with a Channel handle, different than PCAN_NONEBUS, that has not been initialized yet. Each initialized channel should be released when it is not needed anymore.

Initializing a PCAN Channel means:

- to reserve the Channel for the calling application/process.
- to allocate channel resources, like receive and transmit queues.
- to register/connect the Hardware denoted by the channel handle.
- **to check and adapt the bus speed, if the Channel is already in use.** (Only if the Channel was pre-configured as Bitrate Adapting; see: **Bitrate-Adapting Parameter**).
- **to set the channel in Listen-Only mode.** (Only if the channel was pre-configured as Listen-Only; see: **Listen-Only Parameter**).
- to open the messages filter for the application/process.
- to set-up the default values of the different parameters (See **GetValue**).
- **to set the Receive Status of the channel.** (Pre-configured value; see: **Receive Status Parameter**).

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

Take in consideration that initializing a channel causes a reset of the CAN hardware , when the bus status is other than OK. In this way errors like BUSOFF, BUSWARNING, and BUSPASSIVE, are removed.

**PCAN-LAN Channels**

A PCAN-LAN channel doesn't allow changing the bit rate using PCAN-Basic. In order to connect a PCAN-LAN Channel it is
necessary to know the bit rate of the PCAN-Gateway device that is represented by that channel. If the bit rate is not known, the parameter **Bitrate-Adapting** should be used.

**Python Notes**

- Class-Method: Different than the .NET Framework, under Python a variable has to be instantiated with an object of type **PCANBasic** in order to use the API functionality.
- Python's first argument convention: Under Python, ‘self’ is a parameter that is automatically included within the call of this method, within a **PCANBasic** object and hasn't to be indicated in a method call. This parameter represents the calling object itself.

**Example**

The following example shows the initialize and uninitialize processes for a FD capable channel (channel 1 of a PCAN-USB Pro FD hardware). In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

**C#**:

```csharp
string bitrate;
TPCANStatus result;
StringBuilder strMsg;

// Defines a FD Bit rate string with nominal and data Bit rate set to 1 MB
// bitrate = "f_clock_mhz=24, nom_brp=1, nom_tseg1=17,
// nom_tseg2=6, nom_sjw=1, data_brp=1, data_tseg1=16, data_tseg2=7, data_sjw=1"

// The FD capable Channel (PCAN-USB Pro FD) is initialized
// result = PCANBasic.InitializeFD(PCANBasic.PCAN_USBBUS1)
if (result != TPCANStatus.PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    //
```
strMsg = new StringBuilder(256);
PCANBasic.GetErrorText(result, 0, strMsg);
MessageBox.Show(strMsg.ToString());
}
else
    MessageBox.Show("PCAN-USB Pro FD (Ch-1) was initialized\n\n// All initialized channels are released\n// PCANBasic.Uninitialize(PCANBasic.PCAN_NONEBUS);

C++/CLR:

String^ bitrate;
TPCANStatus result;
StringBuilder^ strMsg;

// Defines a FD Bit rate string with nominal and data Bit rate set to 1 MBps
// bitrate = "f_clock_mhz=24, nom_brp=1, nom_tseg1=17, nom_tseg2=6, nom_sjw=1, data_brp=1, data_tseg1=16, data_tseg2=7, data_sjw=1"

// The FD capable Channel (PCAN-USB Pro FD) is initialized
// result = PCANBasic::InitializeFD(PCANBasic::PCAN_USBBUS1)
if (result != TPCANStatus::PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error and show it
    // strMsg = gcnew StringBuilder(256);
    PCANBasic::GetErrorText(result, 0, strMsg);
    MessageBox::Show(strMsg->ToString());
}
else
    MessageBox::Show("PCAN-USB Pro FD (Ch-1) was initialized\n\n// All initialized channels are released\n// PCANBasic::Uninitialize(PCANBasic::PCAN_NONEBUS);

Visual Basic:

```vbnet
Dim bitrate As String
Dim result As TPCANStatus
Dim strMsg As StringBuilder

' Defines a FD Bit rate string with nominal and data Bit rate set to 1 MB
bitrate = "f_clock_mhz=24, nom_brp=1, nom_tseg1=17, nom_tseg2=6, nom_sjw=1, data_brp=1, data_tseg1=16, data_tseg2=7, data_sjw=1"

' The FD capable Channel (PCAN-USB Pro FD) is initialized
result = PCANBasic.InitializeFD(PCANBasic.PCAN_USB)
If result <> TPCANStatus.PCAN_ERROR_OK Then
    ' An error occurred, get a text describing the error
    strMsg = New StringBuilder(256)
    PCANBasic.GetErrorText(result, 0, strMsg)
    MessageBox.Show(strMsg.ToString)
Else
    MessageBox.Show("PCAN-USB Pro FD (Ch-1) was initialized")
End If

' All initialized channels are released
PCANBasic.Uninitialize(PCANBasic.PCAN_NONEBUS)
```

Pascal OO:

```pascal
var
    bitrate: String;
    result : TPCANStatus;
    strMsg: array [0..256] of Char;
begin
    // Defines a FD Bit rate string with nominal and data Bit rate set to 1 MB
    bitrate := 'f_clock_mhz=24, nom_brp=1, nom_tseg1=17, nom_tseg2=6, nom_sjw=1, data_brp=1, data_tseg1=16, data_tseg2=7, data_sjw=1';
```
The FD capable Channel (PCAN-USB Pro FD) is initialized

```c
result := TPCANBasic.InitializeFD(TPCANBasic.PCAN_USBBUS1)
If (result <> PCAN_ERROR_OK) Then
begin
  // An error occurred, get a text describing the error
  TPCANBasic.GetErrorText(result, 0, strMsg)
  MessageBox(0, strMsg, 'Error', MB_OK);
end
else
  MessageBox(0, 'PCAN-USB Pro FD (Ch-1) was initialized'

// All initialized channels are released
//
TPCANBasic.Uninitialize(TPCANBasic.PCAN_NONEBUS)
end;
```

Python:

```python
# Defines a FD Bit rate string with nominal and data Bit rate set to 1 MB
bitrate = "f_clock_mhz=24, nom_brp=1, nom_tseg1=17, nom_tseg2=6, nom_sjw=1, data_brp=1, data_tseg1=16, data_tseg2=7, data_sjw=1"

# The FD capable Channel (PCAN-USB Pro FD) is initialized
objPCAN = PCANBasic()
result = objPCAN.InitializeFD(PCAN_USBBUS1, bitrate)
if result != PCAN_ERROR_OK:
  # An error occurred, get a text describing the error
  result = objPCAN.GetErrorText(result)
  print result[1]
else:
  print "PCAN-USB Pro FD (Ch-1) was initialized"

# All initialized channels are released
```
| objPCAN.Uninitialize(PCAN_NONEBUS) |

### See Also

- Uninitialize
- ReadFD
- WriteFD

**Plain function Version:** [CAN_InitializeFD](#)
Uninitialize

Uninitializes a PCAN Channel.

Syntax

Pascal OO

```pascal
class function Uninitialize(
    Channel: TPCANHandle
): TPCANStatus;
```

C#

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_Uninitialize")
public static extern TPCANStatus Uninitialize(
    [MarshalAs(UnmanagedType.U1)]
    TPCANHandle Channel);
```

C++ / CLR

```cpp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_Uninitialize")
static TPCANStatus Uninitialize(
    [MarshalAs(UnmanagedType::U1)]
    TPCANHandle Channel);
```

Visual Basic

```vbnet
<DllImport("PCANBasic.dll", EntryPoint:="CAN_Uninitialize")>
Public Shared Function Uninitialize( _
    ByVal Channel As TPCANHandle) As TPCANStatus
End Function
```

Python

```python
def Uninitialize(
    self,
```

```python```
**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
</tbody>
</table>

**Returns**

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

| PCAN_ERROR_INITIALIZE: | Indicates that the given PCAN channel cannot be uninitialized because it was not found in the list of reserved channels of the calling application. |

**Remarks**

A PCAN Channel can be released using one of this possibilities:

**Single-Release:** Given a handle of a PCAN 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 PCAN_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.

**Transmit-queue at uninitialize:** When a PCAN-Basic channel connection is terminated, the underlying hardware's transmit-queue will not immediately be discarded. PCAN-Basic will wait some time before finalizing, so that the hardware has time to send (or try to send) those unsent messages. When the time is up (amount 500 milliseconds), the rest of the messages in the queue (if any) are
discarded.

**Python Notes**

- Class-Method: Unlike the .NET Framework, under Python a variable has to be instantiated with an object of type `PCANBasic` in order to use the API functionality.
- Python's first argument convention: Under Python, 'self' is a parameter that is automatically included within the call of this method, within a `PCANBasic` object and hasn't to be indicated in a method call. This parameter represents the calling object itself.

**Example**

The following example shows the initialize and uninitialize (Single-Release) processes for the PCAN_PCIBUS1 channel. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: To see an example of Multiple-Release, see the `Initialize` method.

**C#:**

```csharp
TPCANStatus result;
StringBuilder strMsg;

strMsg = new StringBuilder(256);

// The Plug & Play Channel (PCAN-PCI) is initialize
// result = PCANBasic.Initialize(PCANBasic.PCAN_PCIBUS1)
if (result != TPCANStatus.PCAN_ERROR_OK)
{
    // An error occurred, get a text describing th
    // PCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox.Show(strMsg.ToString());
}
```
else
    MessageBox.Show("PCAN-PCI (Ch-1) was initialized");

    // The PCI Channel is released
    //
    result = PCANBasic.Uninitialize(PCANBasic.PCAN_PCI)

    if (result != TPCANStatus.PCAN_ERROR_OK)
    {
        // An error occurred, get a text describing the error
        //
        PCANBasic.GetErrorText(result, 0, strMsg);
        MessageBox.Show(strMsg.ToString());
    }
else
    MessageBox.Show("PCAN-PCI (Ch-1) was released");

C++/CLR:

```cpp
TPCANStatus result;
StringBuilder^ strMsg;

strMsg = gcnew StringBuilder(256);

// The Plug & Play Channel (PCAN-PCI) is initialized
//
result = PCANBasic::Initialize(PCANBasic::PCAN_PCI);
if (result != TPCANStatus::PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    //
    PCANBasic::GetErrorText(result, 0, strMsg);
    MessageBox::Show(strMsg->ToString());
}
else
    MessageBox::Show("PCAN-PCI (Ch-1) was initialized");
```
// The PCI Channel is released

result = PCANBasic::Uninitialize(PCANBasic::PCAN_PCI
if (result != TPCANStatus::PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    PCANBasic::GetErrorText(result, 0, strMsg);
    MessageBox::Show(strMsg->ToString());
}
else
    MessageBox::Show("PCAN-PCI (Ch-1) was released

Visual Basic:

Dim result As TPCANStatus
Dim strMsg As StringBuilder

strMsg = New StringBuilder(256)

' The Plug & Play Channel (PCAN-PCI) is initialized
result = PCANBasic.Initialize(PCANBasic.PCAN_PCI
If result <> TPCANStatus.PCAN_ERROR_OK Then
    ' An error occurred, get a text describing the error
    PCANBasic.GetErrorText(result, 0, strMsg)
    MessageBox.Show(strMsg.ToString)
Else
    MessageBox.Show("PCAN-PCI (Ch-1) was initialized
End If

' The PCI Channel is released
result = PCANBasic.Uninitialize(PCANBasic.PCAN_PCI)
If result <> TPCANStatus.PCAN_ERROR_OK Then
    ' An error occurred, get a text describing the error
    PCANBasic.GetErrorText(result, 0, strMsg)
    MessageBox.Show(strMsg.ToString)
Else
    MessageBox.Show("PCAN-PCI (Ch-1) was released")
End If

Pascal OO:

var
    result : TPCANStatus;
    strMsg: array [0..256] of Char;
begi
    // The Plug & Play Channel (PCAN-PCI) is initialized
    result := TPCANBasic.Initialize(TPCANBasic.PCAN_PCI);
    If (result <> PCAN_ERROR_OK) Then
        begin
            // An error occurred, get a text describing the error
            TPCANBasic.GetErrorText(result, 0, strMsg);
            MessageBox(0, strMsg, 'Error', MB_OK);
        end
    else
        MessageBox(0, 'PCAN-PCI (Ch-1) was initialized'
    ....

    // The PCI Channel is released
    //
    result := TPCANBasic.Uninitialize(TPCANBasic.PCAN_PCI);
    If (result <> PCAN_ERROR_OK) Then
        begin
            // An error occurred, get a text describing the error
            TPCANBasic.GetErrorText(result, 0, strMsg);
            MessageBox(0, strMsg, 'Error', MB_OK);
        end
    else
        MessageBox(0, 'PCAN-PCI (Ch-1) was released'

TPCANBasic.GetErrorText(result, 0, strMsg);
MessageBox(0, strMsg, 'Error', MB_OK);
end
else
    MessageBox(0, 'PCAN-PCI (Ch-1) was released
end;

Python:

# The Plug & Play Channel (PCAN-PCI) is initialized
#
objPCAN = PCANBasic()
result = objPCAN.Initialize(PCAN_PCIBUS1, PCAN_BAUD_500K)
if result != PCAN_ERROR_OK:
    # An error occurred, get a text describing the
    #
    result = objPCAN.GetErrorText(result)
    print result[1]
else:
    print "PCAN-PCI (Ch-1) was initialized"

....

# The PCI Channel is released
#
result = objPCAN.Uninitialize(PCAN_PCIBUS1)
if result != PCAN_ERROR_OK:
    # An error occurred, get a text describing the
    #
    result = objPCAN.GetErrorText(result)
    print result[1]
else:
    print "PCAN-PCI (Ch-1) was released"

See Also

Initialize
Plain function Version: CAN_Uninitialize
Reset

Resets the receive and transmit queues of a PCAN Channel.

Syntax

<table>
<thead>
<tr>
<th>Pascal OO</th>
<th>C#</th>
<th>C++ / CLR</th>
<th>Visual Basic</th>
<th>Python</th>
</tr>
</thead>
</table>
| class function    | [DllImport("PCANBasic.dll", EntryPoint = "CAN_Reset")]
| Reset(            | public static extern TPCANStatus Reset(    | static TPCANStatus Reset(     |
| Channel: TPCANHandle | [MarshalAs(UnmanagedType.U1)] TPCANHandle Channel); | [MarshalAs(UnmanagedType::U1)] TPCANHandle Channel); |
| ): TPCANStatus;   |                                             |                                 | <DllImport("PCANBasic.dll", EntryPoint:="CAN_Reset")
|                   |                                             |                                 | Public Shared Function Reset( _
|                   |                                             |                                 |     _
|                   |                                             |                                 | <MarshalAs(UnmanagedType::U1)> _
|                   |                                             |                                 | ByVal Channel As TPCANHandle) As TPCANStatus|
|                   |                                             |                                 | End Function               |
|                   |                                             |                                 |                             |
|                   |                                             |                                 | def Reset( self,            |
|                   |                                             |                                 |                             |        |
### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
</tbody>
</table>

### Returns

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

- **PCAN_ERROR_INITIALIZE**: Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.

### Remarks

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

Normally a reset of the CAN Controller is desired when a bus-off occur. In this case an application cannot use the channel to communicate anymore, until the CAN controller is reset. Consider using the PCAN-Basic parameter 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, but only when no more clients are connected to that channel.

### Python Notes

- Class-Method: Unlike the .NET Framework, under Python a
variable has to be instantiated with an object of type **PCANBasic** in order to use the API functionality.

- **Python's first argument convention:** Under Python, 'self' is a parameter that is automatically included within the call of this method, within a **PCANBasic** object and hasn't to be indicated in a method call. This parameter represents the calling object itself.

**Example**

The following example shows the use of the method Reset on the channel **PCAN_PCIBUS1**. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

**C#:**

```csharp
TPCANStatus result;
StringBuilder strMsg;

strMsg = new StringBuilder(256);

......

// The PCI Channel is reset
//
result = PCANBasic.Reset(PCANBasic.PCAN_PCIBUS1);
if (result != TPCANStatus.PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    //
    PCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox.Show(strMsg.ToString());
}
else
    MessageBox.Show("PCAN-PCI (Ch-1) was reset");
```
C++/CLR:

```c++
TPCANStatus result;
StringBuilder^ strMsg;

strMsg = gcnew StringBuilder(256);

......

// The PCI Channel is reset
//
result = PCANBasic::Reset(PCANBasic::PCAN_PCIBUS1);
if (result != TPCANStatus::PCAN_ERROR_OK)
{
  // An error occurred, get a text describing the error
  PCANBasic::GetErrorText(result, 0, strMsg);
  MessageBox::Show(strMsg->ToString());
}
else
  MessageBox::Show("PCAN-PCI (Ch-1) was reset");
```

Visual Basic:

```vbnet
Dim result As TPCANStatus
Dim strMsg As StringBuilder

strMsg = New StringBuilder(256)

......

' The PCI Channel is reset'
result = PCANBasic.Reset(PCANBasic.PCAN_PCIBUS1)
If result <> TPCANStatus.PCAN_ERROR_OK Then
  ' An error occurred, get a text describing the error
  PCANBasic.GetErrorText(result, 0, strMsg)
```
MessageBox.Show(strMsg.ToString)
Else
    MessageBox.Show("PCAN-PCI (Ch-1) was reset")
End If

Pascal OO:

var
    result : TPCANStatus;
    strMsg: array [0..256] of Char;
begin

    ......

    // The PCI Channel is reset
    //
    result := TPCANBasic.Reset(TPCANBasic.PCAN_PCI
    If (result <> PCAN_ERROR_OK) Then
        begin
            // An error occurred, get a text describing
            //
            TPCANBasic.GetErrorText(result, 0, strMsg)
            MessageBox(0, strMsg, 'Error',MB_OK);
        end
    else
        MessageBox(0,'PCAN-PCI (Ch-1) was reset','

Python:

    ......

    # The PCI Channel is released
    #
    result = objPCAN.Reset(PCAN_PCIBUS1)
    if result != PCAN_ERROR_OK:
        # An error occurred, get a text describing the
        #
        result = objPCAN.GetErrorText(result)
print result[1]
else:
    print "PCAN-PCI (Ch-1) was reset"

See Also

Read
Write
SetValue

Plain function Version: CAN_Reset
GetStatus

Gets the current BUS status of a PCAN Channel.

Syntax

**Pascal OO**

```pascal
class function GetStatus(
    Channel: TPCANHandle
): TPCANStatus;
```

**C#**

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_GetStatus")
public static extern TPCANStatus GetStatus(
    [MarshalAs(UnmanagedType.U1)]
    TPCANHandle Channel);
```

**C++ / CLR**

```cpp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_GetStatus")
static TPCANStatus GetStatus(
    [MarshalAs(UnmanagedType::U1)]
    TPCANHandle Channel);
```

**Visual Basic**

```vbnet
<DllImport("PCANBasic.dll", EntryPoint:="CAN_GetStatus")>
Public Shared Function GetStatus( _
    ByVal Channel As TPCANHandle) As TPCANStatus
End Function
```

**Python**

```python
def GetStatus(  
    self,
```
**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see <a href="#">TPCANHandle</a>).</td>
</tr>
</tbody>
</table>

**Returns**

The return value is a [TPCANStatus](#) code. The typical return values are:

<table>
<thead>
<tr>
<th>PCAN_ERROR_INITIALIZE:</th>
<th>Indicates that the given PCAN Channel was not found in the list of initialized channels of the calling application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_BUSLIGHT:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-light status.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSHEAVY:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-heavy status.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSOFF:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-off status.</td>
</tr>
<tr>
<td>PCAN_ERROR_OK:</td>
<td>Indicates that the status of the given PCAN Channel is OK.</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.

**Python Notes**

- **Class-Method:** Unlike the .NET Framework, under Python a variable has to be instantiated with an object of type `PCANBasic` in order to use the API functionality.
- **Python’s first argument convention:** Under Python, ‘self’ is a parameter that is automatically included within the call of this method, within a `PCANBasic` object and hasn't to be indicated in a method call. This parameter represents the calling object itself.

**Example**

The following example shows the use of the method `GetStatus` on the channel `PCAN_PCIBUS1`. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

**C#:**

```csharp
TPCANStatus result;
StringBuilder strMsg;

strMsg = new StringBuilder(256);

......

// Check the status of the PCI Channel
//
result = PCANBasic.GetStatus(PCANBasic.PCAN_PCIBUS
switch (result)```
```csharp
{ 
    case TPCANStatus.PCAN_ERROR_BUSLIGHT:
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-LIGHT status..."
        break;
    case TPCANStatus.PCAN_ERROR_BUSHEAVY:
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-HEAVY status..."
        break;
    case TPCANStatus.PCAN_ERROR_BUSOFF:
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-OFF status..."
        break;
    case TPCANStatus.PCAN_ERROR_OK:
        MessageBox.Show("PCAN-PCI (Ch-1): Status is OK"
        break;
    default:
        // An error occurred, get a text describing the error
        // PCANBasic.GetErrorText(result, 0, strMsg);
        MessageBox.Show(strMsg.ToString());
        break;
}

C++/CLR:

TPCANStatus result;
StringBuilder^ strMsg;

strMsg = gcnew StringBuilder(256);

......

// Check the status of the PCI Channel
// result = PCANBasic::GetStatus(PCANBasic::PCAN_PCIBUS1
switch (result) {
    case TPCANStatus::PCAN_ERROR_BUSLIGHT:
        MessageBox::Show("PCAN-PCI (Ch-1): Handling a BUS-LIGHT status..."
        break;
```
case TPCANStatus::PCAN_ERROR_BUSHEAVY:
    MessageBox::Show("PCAN-PCI (Ch-1): Handling a BUS-HEAVY status..."
    break;

case TPCANStatus::PCAN_ERROR_BUSOFF:
    MessageBox::Show("PCAN-PCI (Ch-1): Handling a BUS-OFF status..."
    break;

case TPCANStatus::PCAN_ERROR_OK:
    MessageBox::Show("PCAN-PCI (Ch-1): Status is OK"
    break;

default:
    // An error occurred, get a text describing the error and show it
    //
    PCANBasic::GetErrorText(result, 0, strMsg)
    MessageBox::Show(strMsg->ToString());
    break;
}

Visual Basic:

Dim result As TPCANStatus
Dim strMsg As StringBuilder

strMsg = New StringBuilder(256)

......

' Check the status of the PCI Channel 
' result = PCANBasic.GetStatus(PCANBasic.PCAN_PCIEBUS
Select Case result
    Case TPCANStatus.PCAN_ERROR_BUSLIGHT
        MessageBox.Show("PCAN-PCI (Ch-1): Handling"
    Case TPCANStatus.PCAN_ERROR_BUSHEAVY
        MessageBox.Show("PCAN-PCI (Ch-1): Handling"
    Case TPCANStatus.PCAN_ERROR_BUSOFF
        MessageBox.Show("PCAN-PCI (Ch-1): Handling"
    Case TPCANStatus.PCAN_ERROR_OK
        MessageBox.Show("PCAN-PCI (Ch-1): Status is 

Case Else
    ' An error occurred, get a text describing
    '
    PCANBasic.GetErrorText(result, 0, strMsg)
    MessageBox.Show(strMsg.ToString)
End Select

Pascal OO:

var
    result : TPCANStatus;
    strMsg: array [0..256] of Char;
begin

    ......

    // Check the status of the PCI Channel
    //
    result := TPCANBasic.GetStatus(TPCANBasic.PCAN_PCI
    case result of
        PCAN_ERROR_BUSLIGHT:
            MessageBox(0, 'PCAN-PCI (Ch-1): Handling a
        PCAN_ERROR_BUSHEAVY:
            MessageBox(0, 'PCAN-PCI (Ch-1): Handling a
        PCAN_ERROR_BUSOFF:
            MessageBox(0, 'PCAN-PCI (Ch-1): Handling a
        PCAN_ERROR_OK:
            MessageBox(0, 'PCAN-PCI (Ch-1): Status is O
    else
    begin
        // An error occurred, get a text describing
        //
        TPCANBasic.GetErrorText(result, 0, strMsg)
        MessageBox(0, strMsg, 'Error',MB_OK);
    end;
end;

Python:
# Check the status of the PCI Channel

```python
result = objPCAN.GetStatus(PCAN_PCIBUS1)
if result == PCAN_ERROR_BUSLIGHT:
    print "PCAN-PCI (Ch-1): Handling a BUS-LIGHT status..."
elif result == PCAN_ERROR_BUSHEAVY:
    print "PCAN-PCI (Ch-1): Handling a BUS-HEAVY status..."
elif result == PCAN_ERROR_BUSOFF:
    print "PCAN-PCI (Ch-1): Handling a BUS-OFF status..."
elif result == PCAN_ERROR_OK:
    print "PCAN-PCI (Ch-1): Status is OK"
else:
    # An error occurred, get a text describing the error
    result = objPCAN.GetErrorText(result)
    print result[1]
```

See Also

- Parameter Value Definitions
- TPCANParameter

Plain function Version: CAN_GetStatus
Read

Reads a CAN message from the receive queue of a PCAN Channel.

## Overloads

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Read(TPCANHandle, TPCANMsg)</code></td>
<td>Reads a CAN message from the receive queue.</td>
</tr>
<tr>
<td><code>Read(TPCANHandle, TPCANMsg, TPCANTimestamp)</code></td>
<td>Reads a CAN message and its timestamp from the receive queue.</td>
</tr>
<tr>
<td><code>Read(TPCANHandle)</code></td>
<td>Reads a CAN message and its timestamp from the receive queue.</td>
</tr>
</tbody>
</table>
Read(TPCANHandle, TPCANMsg)

Reads a CAN message from the receive queue of a PCAN Channel.

Syntax

### Pascal OO

```pascal
class function Read(
    Channel: TPCANHandle;
    var MessageBuffer: TPCANMsg
): TPCANStatus; overload;
```

### C#

```csharp
public static TPCANStatus Read(
    TPCANHandle Channel,
    out TPCANMsg MessageBuffer);
```

### C++ / CLR

```cpp
static TPCANStatus Read(
    TPCANHandle Channel,
    TPCANMsg %MessageBuffer);
```

### Visual Basic

```vbnet
Public Shared Function Read( _
    ByVal Channel As TPCANHandle, _
    ByRef MessageBuffer As TPCANMsg) As TPCANStatus
End Function
```

### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
</tbody>
</table>
MessageBuffer | A `TPCANMsg` buffer to store the CAN message.

### Returns

The return value is a `TPCANStatus` code. `PCAN_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>PCAN_ERROR_I LLPARAMVAL:</td>
<td>Indicates that the parameters passed to the method are invalid. Check the value of the MessageBuffer; it should point to a <code>TPCANMsg</code> structure.</td>
</tr>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSLIGHT:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-light status.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSHEAVY:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-heavy status.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSOFF:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-off status.</td>
</tr>
<tr>
<td>PCAN_ERROR_QRCVEMPTY:</td>
<td>Indicates that the receive queue of the Channel is empty.</td>
</tr>
</tbody>
</table>

### Remarks
The Read method returns received messages or status messages from the receive queue. It is important to call Read repeatedly until the queue is empty. In case there are no more messages in queue, the value PCAN_ERROR_QRCVEMPTY is returned. The error code PCAN_ERROR_QRCVEMPTY is also returned if the reception of messages is disabled. See Receive Status Parameter for more information.

The receive queue can contain up to 32767 messages.

If the time when the message was received is needed, use the overloaded Read method.

There are two possibilities for reading messages from the receive queue of a Channel:

**Time-Triggered Reading:** Consists in periodically calls to the Read method. Typically, an application start a timer that every 50 or 100 milliseconds check for messages, calling the Read method in a loop until the value of PCAN_ERROR_QRCVEMPTY or another error condition is reached.

**Event-Triggered Reading:** Consists in reacting to a notification sent by the PCAN driver to a registered application, when a message is received and inserted in its receive queue. See Using Events to obtain more information about reading with events.

**About bus errors / Status messages**

If a bus-off error occur, an application cannot use the channel to communicate anymore, until the CAN controller is reset. With PCAN-Basic it is not possible to reset the CAN controller through a method directly. 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 BUSOFF, BUSHEAVY, and BUSLIGHT, is to uninitialize and initialise again the channel used. This causes a hardware reset, but only when no more clients are connected to that channel.
The message type (see TPCANMessageType) of a CAN message indicates if the message is a 11-bit, 29-bit, RTR, Error, or Status message. This value should be checked every time a message has been read successfully.

If the bit PCAN_MESSAGE_ERRFRAME is set in the TPCANMsg.MSGTYPE field, the message is an Error frame (see Error Frames).

If the bit PCAN_MESSAGE_STATUS is set in the TPCANMsg.MSGTYPE field, the message is a Status message. The ID and LEN fields do not contain valid data. The first 4 data bytes of the message contain the Error Code. The MSB of the Error Code is in data byte 0, the LSB is in data byte 3. If a status message was read the return value of Read is also the error code.

Examples:

<table>
<thead>
<tr>
<th>Data0</th>
<th>Data1</th>
<th>Data2</th>
<th>Data3</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>02h</td>
<td>PCAN_ERROR_OVERRUN</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>04h</td>
<td>PCAN_ERROR_BUSLIGHT</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>08h</td>
<td>PCAN_ERROR_BUSHEAVY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

The following example shows the use of method Read on the channel PCAN_USB BUS1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: It is assumed that the channel was already initialized and that the following code is an OnTimer event handler method.

C#:

```csharp
TPCANStatus result;
StringBuilder strMsg;
TPCANMsg msg;

strMsg = new StringBuilder(256);

do
{
    // Check the receive queue for new messages
    //
    result = PCANBasic.Read(PCANBasic.PCAN_USB BUS1
    if (result != TPCANStatus.PCAN_ERROR_QRCVEMPTY
    {
        // Process the received message
        //
        MessageBox.Show("A message was received");
        ProcessMessage(msg);
    }
    else
```
{ // An error occurred, get a text describing the error and show it
    PCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox.Show(strMsg.ToString());
    // Here can be decided if the loop has to terminate (e.g. status is bus-off)
    // HandleReadError(result);
}
// Try to read a message from the receive queue of PCAN-USB, Channel 1,
// until the queue is empty
//}
}while((result & TPCANStatus.PCAN_ERROR_QRCVEMPTY)

C++CLR:

```cpp
TPCANStatus result;
StringBuilder^ strMsg;
TPCANMsg msg;

strMsg = gcnew StringBuilder(256);

do
{
    // Check the receive queue for new messages
    result = PCANBasic::Read(PCANBasic::PCAN_USBBUS1
        if (result != TPCANStatus::PCAN_ERROR_QRCVEMPTY
            {
                // Process the received message
                //
                MessageBox::Show("A message was received")
                ProcessMessage(msg);
            }
        else
            {
                // An error occurred, get a text describing
```
//
PCANBasic::GetErrorText(result, 0, strMsg)
MessageBox::Show(strMsg->ToString());
// Here can be decided if the loop has to
// status is bus-off)
//
HandleReadError(result);
}
// Try to read a message from the receive queue of
// until the queue is empty
//
}while((result & TPCANStatus::PCAN_ERROR_QRCVEMPTY)

Visual Basic:
Dim result As TPCANStatus
Dim strMsg As StringBuilder
Dim msg As TPCANMsg
strMsg = New StringBuilder(256)

Do
' Check the receive queue for new messages
'  
result = PCANBasic.Read(PCANBasic.PCAN_USBBUS1
If result <> TPCANStatus.PCAN_ERROR_QRCVEMPTY
  MessageBox.Show("A message was received")
  ProcessMessage(msg)
Else
' An error occurred, get a text describing
'  
  PCANBasic.GetErrorText(result, 0, strMsg)
  MessageBox.Show(strMsg.ToString())
  ' Here can be decided if the loop has to be
  ' status is bus-off)
  '  
  HandleReadError(result)
End If
Try to read a message from the receive queue of PCAN-USB, Channel 1, until the queue is empty.

Loop While ((result And TPCANStatus.PCAN_ERROR_QRCVEMPTY) <> 0)

Pascal OO:

```
var
  result : TPCANStatus;
  strMsg: array [0..256] of Char;
  msg: TPCANMsg;
begin
  repeat
    // Check the receive queue for new message
    //
    result := TPCANBasic.Read(TPCANBasic.PCAN_USB_BUS1);
    If (result <> PCAN_ERROR_QRCVEMPTY) Then begin
      // Process the received message
      //
      MessageBox(0,'A message was received', ProcessMessage(msg));
    end
    else begin
      // An error occurred, get a text description
      //
      TPCANBasic.GetErrorText(result, 0, strMsg);
      MessageBox(0, strMsg, 'Error', MB_OK);
      // Here can be decided if the loop has to be terminated (eg. bus-off)
      // status is bus-off)
      //
      HandleReadError(result);
    end;
  // Try to read a message from the receive queue
  // until the queue is empty
  //
  until ((TPCANStatus(Integer(result) AND TPCANStatus.PCAN_ERROR_QRCVEMPTY)) <> 0)
```
See Also

Write
Using Events
Error Frames

Plain function Version: CAN_Read
Read(TPCANHandle, TPCANMsg, TPCANTimestamp)

Reads a CAN message and its time stamp from the receive queue of a PCAN Channel.

**Syntax**

<table>
<thead>
<tr>
<th>Pascal OO</th>
<th>C#</th>
<th>C++ / CLR</th>
<th>Visual Basic</th>
</tr>
</thead>
<tbody>
<tr>
<td>class function Read(</td>
<td>[DllImport(&quot;PCANBasic.dll&quot;,</td>
<td>[DllImport(&quot;PCANBasic.dll&quot;,</td>
<td>&lt;DllImport(&quot;PCANBasic.dll&quot;,</td>
</tr>
<tr>
<td>Channel: TPCANHandle;</td>
<td>EntryPoint = &quot;CAN_Read&quot;,</td>
<td>EntryPoint = &quot;CAN_Read&quot;,</td>
<td>EntryPoint:=&quot;CAN_Read&quot;,</td>
</tr>
<tr>
<td>var MessageBuffer: TPCANMsg;</td>
<td>public static extern TPCANStatus Read(</td>
<td>public static extern TPCANStatus Read(</td>
<td>&lt;DllImport(&quot;PCANBasic.dll&quot;,</td>
</tr>
<tr>
<td>var TimestampBuffer: TPCANTimestamp</td>
<td>[MarshalAs(UnmanagedType.U1)] TPCANHandle Channel,</td>
<td>[MarshalAs(UnmanagedType::U1)] TPCANHandle Channel,</td>
<td>EntryPoint:=&quot;CAN_Reader&quot;,</td>
</tr>
<tr>
<td>): TPCANStatus; overload;</td>
<td>out TPCANMsg MessageBuffer,</td>
<td>out TPCANMsg %MessageBuffer,</td>
<td>) = &quot;CAN_Reader&quot;,</td>
</tr>
<tr>
<td></td>
<td>ref TPCANTimestamp TimestampBuffer);</td>
<td>ref TPCANTimestamp %TimestampBuffer);</td>
<td>) = &quot;CAN_Reader&quot;,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>) = &quot;CAN_Reader&quot;,</td>
</tr>
</tbody>
</table>

Read(TPCANHandle, TPCANMsg, TPCANTimestamp)
Public Shared Function Read(_
    <MarshalAs(UnmanagedType.U1)> _
    ByVal Channel As TPCANHandle, _
    ByRef MessageBuffer As TPCANMsg, _
    ByRef TimestampBuffer As TPCANTimestamp) As _
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>MessageBuffer</td>
<td>A TPCANMsg buffer to store the CAN message.</td>
</tr>
<tr>
<td>TimestampBuffer</td>
<td>A TPCANTimestamp buffer to get the reception time of the message.</td>
</tr>
</tbody>
</table>

Returns

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

<table>
<thead>
<tr>
<th>PCAN_ERROR_I LLPARAMVAL:</th>
<th>Indicates that the parameters passed to the method are invalid. Check the value of the MessageBuffer; it should point to a TPCANMsg structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_INITIALIZ E:</td>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
</tbody>
</table>
### Remarks

The [Read](#) method returns received messages or status messages from the receive queue. It is important to call [Read](#) repeatedly until the queue is empty. In case there are no more messages in queue, the value `PCAN_ERROR_QRCVEMPTY` is returned. The error code `PCAN_ERROR_QRCVEMPTY` is also returned if the reception of messages is disabled. See [Receive Status Parameter](#) for more information.

The receive queue can contain up to 32767 messages.

If the time when the message was received is not needed, use the overloaded [Read](#) method.

There are two possibilities for reading messages from the receive queue of a Channel:

**Time-Triggered Reading**: Consists in periodically calls to the [Read](#) method. Typically, an application start a timer that every 50 or 100 milliseconds check for messages, calling the [Read](#) method in a loop until the value of `PCAN_ERROR_QRCVEMPTY` or another error condition is reached.
Event-Triggered Reading: Consists in reacting to a notification sent by the PCAN driver to a registered application, when a message is received and inserted in its receive queue. See Using Events to obtain more information about reading with events.

About bus errors / Status messages

If a bus-off error occur, an application cannot use the channel to communicate anymore, until the CAN controller is reset. With PCAN-Basic it is not possible to reset the CAN controller through a method directly. 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 BUSOFF, BUSHEAVY, and BUSLIGHT, is to uninitialize and initialise again the channel used. This causes a hardware reset, but only when no more clients are connected to that channel.

The message type (see TPCANMessageType) of a CAN message indicates if the message is a 11-bit, 29-bit, RTR, Error, or Status message. This value should be checked every time a message has been read successfully.

If the bit PCAN_MESSAGE_ERRFRAME is set in the TPCANMsg.MSGTYPE field, the message is an Error frame (see Error Frames).

If the bit PCAN_MESSAGE_STATUS is set in the TPCANMsg.MSGTYPE field, the message is a Status message. The ID and LEN fields do not contain valid data. The first 4 data bytes of the message contain the Error Code. The MSB of the Error Code is in data byte 0, the LSB is in data byte 3. If a status message was read the return value of Read is also the error code.

Examples:

<table>
<thead>
<tr>
<th>Data0</th>
<th>Data1</th>
<th>Data2</th>
<th>Data3</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>02h</td>
<td>PCAN_ERROR_OVERRUN</td>
</tr>
</tbody>
</table>
Example

The following example shows the use of method `Read` on the channel PCAN_USBBUS1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: It is assumed that the channel was already initialized and that the following code is an OnTimer event handler method.

C#:

```csharp
TPCANStatus result;
StringBuilder strMsg;
TPCANMsg msg;
```
TPCANTimestamp time;

strMsg = new StringBuilder(256);

do {
    // Check the receive queue for new messages
    //
    result = PCANBasic.Read(PCANBasic.PCAN_USBBUS1)
    if (result != TPCANStatus.PCAN_ERROR_QRCVEMPTY) {
        // Process the received message
        //
        MessageBox.Show("A message was received");
        ProcessMessage(msg, time);
    }
    else {
        // An error occurred, get a text describing
        //
        PCANBasic.GetErrorText(result, 0, strMsg);
        MessageBox.Show(strMsg.ToString());
        // Here can be decided if the loop has to
        // status is bus-off
        //
        HandleReadError(result);
    }
    // Try to read a message from the receive queue of
    // until the queue is empty
    //
} while ((result & TPCANStatus.PCAN_ERROR_QRCVEMPTY))

C++CLR:

TPCANStatus result;
StringBuilder^ strMsg;
TPCANMsg msg;
TPCANTimestamp time;
strMsg = gcnew StringBuilder(256);

do {
    // Check the receive queue for new messages
    //
    result = PCANBasic::Read(PCANBasic::PCAN_USBBUS1
    if (result != TPCANStatus::PCAN_ERROR_QRCVEMPTY
    {
        // Process the received message
        //
        MessageBox::Show("A message was received")
        ProcessMessage(msg, time);
    }
    else
    {
        // An error occurred, get a text describing the error
        //
        PCANBasic::GetErrorText(result, 0, strMsg)
        MessageBox::Show(strMsg->ToString());
        // Here can be decided if the loop has to be terminated (e.g. bus-off)
        //
        HandleReadError(result);
    }
    // Try to read a message from the receive queue of the PCAN-USB, Channel 1,
    // until the queue is empty
    //
}while((result & TPCANStatus::PCAN_ERROR_QRCVEMPTY)

Visual Basic:

Dim result As TPCANStatus
Dim strMsg As StringBuilder
Dim msg As TPCANMsg
Dim time As TPCANTimestamp
```csharp
strMsg = New StringBuilder(256)

Do
    ' Check the receive queue for new messages
    result = PCANBasic.Read(PCANBasic(PCAN_USBBUS1)
    If result <> TPCANStatus.PCAN_ERROR_QRCVEMPTY
        MessageBox.Show("A message was received")
        ProcessMessage(msg, time)
    Else
        ' An error occurred, get a text describing
        PCANBasic.GetErrorText(result, 0, strMsg)
        MessageBox.Show(strMsg.ToString())
        ' Here can be decided if the loop has to be
        ' status is bus-off)
        HandleReadError(result);
    End If
    ' Try to read a message from the receive queue of
    ' until the queue is empty
    Loop While ((result And TPCANStatus.PCAN_ERROR_QRCVEMPTY)
```

### Pascal OO:

```pascal
var
    result : TPCANStatus;
    strMsg: array [0..256] of Char;
    msg: TPCANMsg;
    time: TPCANTimestamp;
begin
    repeat
        // Check the receive queue for new message
        //
        result := TPCANBasic.Read(TPCANBasic.PCAN_USBBUS1);
        If (result <> PCAN_ERROR_QRCVEMPTY) Then begin
```
// Process the received message
//
MessageBox(0,'A message was received', ProcessMessage(msg, time);
end
else
begin

// An error occurred, get a text describing the error
//
TPCANBasic.GetErrorText(result, 0, strMsg);
MessageBox(0, strMsg, 'Error', MB_OK);

// Here can be decided if the loop has to be terminated (e.g. the bus status is bus-off)
//
HandleReadError(result);
end;

// Try to read a message from the receive queue
// until the queue is empty
//
until ((TPCANStatus(Integer(result) AND Integer(result)) AND Integer(result))

See Also

Write
Using Events
Error Frames

Plain function Version: CAN_Read
Read(TPCANHandle)

Reads a CAN message and its time stamp from the receive queue of a PCAN Channel.

Syntax

```python
def Read(self, Channel)
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
</tbody>
</table>

Returns

The return value is a 3-touple. The order of the returned values is as follow:

[0]: The method's return value as a TPCANStatus code. PCAN_ERROR_OK is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>PCAN_ERROR_ILLPARAMVAL:</th>
<th>Indicates that the parameters passed to the method are invalid. Check the value of the MessageBuffer; it should point to a TPCANMsg structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the given PCAN channel was not found in the</td>
</tr>
</tbody>
</table>
list of initialized channels of the calling application.

<table>
<thead>
<tr>
<th>PCAN_ERROR_BUSLIGHT:</th>
<th>Indicates a bus error within the given PCAN Channel. The hardware is in bus-light status.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_BUSHEAVY:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-heavy status.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSOFF:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-off status.</td>
</tr>
<tr>
<td>PCAN_ERROR_QRCVEMPTY:</td>
<td>Indicates that the receive queue of the Channel is empty.</td>
</tr>
</tbody>
</table>

[1]: A TPCANMsg structure with the CAN message read.

[2]: A TPCANTimestamp structure with the time when a message was read.

Remarks

The Read method returns received messages or status messages from the receive queue. It is important to call Read repeatedly until the queue is empty. In case there are no more messages in queue, the value PCAN_ERROR_QRCVEMPTY is returned. The error code PCAN_ERROR_QRCVEMPTY is also returned if the reception of messages is disabled. See Receive Status Parameter for more information.

The receive queue can contain up to 32767 messages.

There are two possibilities for reading messages from the receive queue of a Channel:

Time-Triggered Reading: Consists in periodically calls to the Read
method. Typically, an application start a timer that every 50 or 100 milliseconds check for messages, calling the Read method in a loop until the value of PCAN_ERROR_QRCVEMTY or another error condition is reached.

**Event-Triggered Reading**: Consists in reacting to a notification sent by the PCAN driver to a registered application, when a message is received and inserted in its receive queue. See Using Events to obtain more information about reading with events.

**About bus errors / Status messages**

If a bus-off error occur, an application cannot use the channel to communicate anymore, until the CAN controller is reset. With PCAN-Basic it is not possible to reset the CAN controller through a method directly. 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 BUSOFF, BUSHEAVY, and BUSLIGTH, is to uninitialize and initialise again the channel used. This causes a hardware reset, but only when no more clients are connected to that channel.

The message type (see TPCANMessageType) of a CAN message indicates if the message is a 11-bit, 29-bit, RTR, Error, or Status message. This value should be checked every time a message has been read successfully.

If the bit PCAN_MESSAGE_ERRFRAME is set in the TPCANMsg.MSGTYPE field, the message is an Error frame (see Error Frames).

If the bit PCAN_MESSAGE_STATUS is set in the TPCANMsg.MSGTYPE field, the message is a Status message. The ID and LEN fields do not contain valid data. The first 4 data bytes of the message contain the Error Code. The MSB of the Error Code is in data byte 0, the LSB is in data byte 3. If a status message was read the return value of Read is also the error code.

Examples:
<table>
<thead>
<tr>
<th>Data0</th>
<th>Data1</th>
<th>Data2</th>
<th>Data3</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>02h</td>
<td>PCAN_ERROR_OVERRUN</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>04h</td>
<td>PCAN_ERROR_BUSLIGHT</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>08h</td>
<td>PCAN_ERROR_BUSHEAVY</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>10h</td>
<td>PCAN_ERROR_BUSOFF</td>
</tr>
</tbody>
</table>

**Python Notes**

- **Class-Method**: Unlike the .NET Framework, under Python a variable has to be instantiated with an object of type [PCANBasic](#) in order to use the API functionality.
- **Python’s first argument convention**: Under Python, ‘self’ is a parameter that is automatically included within the call of this method, within a [PCANBasic](#) object and hasn't to be indicated in a method call. This parameter represents the calling object itself.

**Example**
The following example shows the use of method `Read` on the channel PCAN_USBBUS1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: It is assumed that the channel was already initialized and that the following code is executed periodically.

**Python:**

```python
readResult = PCAN_ERROR_OK,
while (readResult[0] & PCAN_ERROR_QRCVEMPTY) != PCAN_ERROR_OK:
    # Check the receive queue for new messages
    readResult = objPCAN.Read(PCAN_USBBUS1)
    if readResult[0] != PCAN_ERROR_QRCVEMPTY:
        # Process the received message
        print "A message was received"
        ProcessMessage(result[1],result[2]) # Possible
    else:
        # An error occurred, get a text describing the error and show it
        result = objPCAN.GetErrorText(readResult[0])
        print result[1]
        HandleReadError(readResult[0]) # Possible
```

See Also

- Write
- Using Events
- Error Frames

Plain function Version: `CAN_Read`
ReadFD

Reads a CAN message from the receive queue of a FD capable PCAN Channel.

### Overloads

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ReadFD(TPCANHandle, TPCANMsgFD)</code></td>
<td>Reads a CAN message from the receive queue.</td>
</tr>
<tr>
<td><code>ReadFD(TPCANHandle, TPCANMsgFD, TPCANTimestampFD)</code></td>
<td>Reads a CAN message and its time stamp from the receive queue.</td>
</tr>
<tr>
<td><code>ReadFD(TPCANHandle)</code></td>
<td>Reads a CAN message and its time stamp from the receive queue.</td>
</tr>
</tbody>
</table>
Reads a CAN message from the receive queue of a FD capable PCAN Channel.

Syntax

### Pascal OO

```pascal
class function ReadFD(
    Channel: TPCANHandle;
    var MessageBuffer: TPCANMsgFD
): TPCANStatus; overload;
```

### C#

```csharp
public static TPCANStatus ReadFD(
    TPCANHandle Channel,
    out TPCANMsgFD MessageBuffer
)
```

### C++ / CLR

```cpp
static TPCANStatus ReadFD(
    TPCANHandle Channel,
    TPCANMsgFD %MessageBuffer
);
```

### Visual Basic

```vbnet
Public Shared Function ReadFD(_
    ByVal Channel As TPCANHandle, _
    ByRef MessageBuffer As TPCANMsgFD) As TPCANStatus
End Function
```

### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a FD capable</td>
</tr>
<tr>
<td>PCAN Channel (see TPCANHandle).</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td></td>
</tr>
<tr>
<td>MessageBuffer</td>
<td></td>
</tr>
<tr>
<td>A TPCANMsgFD buffer to store the CAN message.</td>
<td></td>
</tr>
</tbody>
</table>

**Returns**

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

<table>
<thead>
<tr>
<th>PCAN_ERROR_IILLPARAMVAL:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicates that the parameters passed to the method are invalid. Check the value of the MessageBuffer; it should point to a TPCANMsgFD structure.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PCAN_ERROR_ILLOPERATION:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicates that the PCAN Channel passed to the method was not initialized using InitializeFD (plain function: CAN_InitializeFD).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PCAN_ERROR_INITIALIZE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PCAN_ERROR_BUSWARNING:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-warning status.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PCAN_ERROR_BUSPASSIVE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-passive status.</td>
</tr>
</tbody>
</table>
**PCAN_ERROR_BUSOFF:** Indicates a bus error within the given PCAN Channel. The hardware is in bus-off status.

**PCAN_ERROR_QRCVEMPTY:** Indicates that the receive queue of the Channel is empty.

---

**Remarks**

The use of [Read](#) and [ReadFD](#) are mutually exclusive. The PCAN Channel passed to this method must be initialized using [InitializeFD](#) (plain function: [CAN_InitializeFD](#)). Otherwise the error PCAN_ERROR_ILLOPERATION is returned.

If the time when the message was received is needed, use the overloaded [ReadFD](#) method.

The [ReadFD](#) method returns received messages or status messages from the receive queue. It is important to call [ReadFD](#) repeatedly until the queue is empty. In case there are no more messages in queue, the value PCAN_ERROR_QRCVEMPTY is returned. The error code PCAN_ERROR_QRCVEMPTY is also returned if the reception of messages is disabled. See [Receive Status Parameter](#) for more information.

The receive queue can contain up to **32767** messages.

There are two possibilities for reading messages from the receive queue of a Channel:

*Time-Triggered Reading:* Consists in periodically calls to the [ReadFD](#) method. Typically, an application start a timer that every 50 or 100 milliseconds check for messages, calling the [ReadFD](#) method in a loop until the value of PCAN_ERROR_QRCVEMTY or another error condition is reached.

*Event-Triggered Reading:* Consists in reacting to a notification sent by the PCAN driver to a registered application, when a message is
received and inserted in its receive queue. See Using Events to obtain more information about reading with events.

**About bus errors / Status messages**

If a bus-off error occur, an application cannot use the channel to communicate anymore, until the CAN controller is reset. With PCAN-Basic it is not possible to reset the CAN controller through a function directly. 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 BUSOFF, BUSWARNING, and BUSPASSIVE, is to **uninitialize** and **initialize** again the channel used. This causes a hardware reset.

The message type (see `TPCANMessageType`) of a CAN message indicates if the message is a 11-bit, 29-bit, FD, RTR, Error, or Status message. This value should be checked every time a message has been read successfully.

If the bit `PCAN_MESSAGE_ERRFRAME` is set in the `TPCANMsg.MSGTYPE` field, the message is an Error frame (see Error Frames).

If the bit `PCAN_MESSAGE_STATUS` is set in the `TPCANMsg.MSGTYPE` field, the message is a Status message. The ID and DLC fields do not contain valid data. The first 4 data bytes of the message contain the Error Code. The MSB of the Error Code is in data byte 0, the LSB is in data byte 3. If a status message was read the return value of `ReadFD` is also the error code.

Examples:

<table>
<thead>
<tr>
<th>Data0</th>
<th>Data1</th>
<th>Data2</th>
<th>Data3</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>02h</td>
<td>PCAN_ERROR_OVERRUN</td>
</tr>
</tbody>
</table>
Example

The following example shows the use of method `ReadFD` on the channel PCAN_USB1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: It is assumed that the channel was already initialized using the method `InitializeFD` and that the following code is an OnTimer event handler method.

C#:

```csharp
TPCANStatus result;
StringBuilder strMsg;
TPCANMsgFD msg;

strMsg = new StringBuilder(256);
```
do
{
    // Check the receive queue for new messages
    //
    result = PCANBasic.ReadFD(PCANBasic.PCAN_USB_BUS1);
    if (result != TPCANStatus.PCAN_ERROR_QRCVEMPTY)
    {
        // Process the received message
        //
        MessageBox.Show("A message was received");
        ProcessMessage(msg);
    } else
    {
        // An error occurred, get a text describing the error and show it
        //
        PCANBasic.GetErrorText(result, 0, strMsg);
        MessageBox.Show(strMsg.ToString());
        // Here can be decided if the loop has to be terminated (e.g. the bus status is bus-off)
        //
        HandleReadError(result);
    }
    // Try to read a message from the receive queue of
    // until the queue is empty
    //
}while((result & TPCANStatus.PCAN_ERROR_QRCVEMPTY))

C++CLR:

TPCANStatus result;
StringBuilder^ strMsg;
TPCANMsgFD msg;

strMsg = gcnew StringBuilder(256);

do
{  
    // Check the receive queue for new messages  
    //  
    result = PCANBasic::ReadFD(PCANBasic::PCAN_USB)  
    if (result != TPCANStatus::PCAN_ERROR_QRCVEMPTY)  
    {  
        // Process the received message  
        //  
        MessageBox::Show("A message was received")  
        ProcessMessage(msg);  
    }  
    else  
    {  
        // An error occurred, get a text describing the error  
        //  
        PCANBasic::GetErrorText(result, 0, strMsg)  
        MessageBox::Show(strMsg->ToString());  
        // Here can be decided if the loop has to be terminated (e.g. the bus status is bus-off)  
        //  
        HandleReadError(result);  
    }  
    // Try to read a message from the receive queue of the PCAN-USB, Channel 1,  
    // until the queue is empty  
    //  
} while ((result & TPCANStatus::PCAN_ERROR_QRCVEMPTY)

Visual Basic:

Dim result As TPCANStatus  
Dim strMsg As StringBuilder  
Dim msg As TPCANMsgFD

strMsg = New StringBuilder(256)

Do  
   ' Check the receive queue for new messages  
   '  

result = PCANBasic.ReadFD(PCANBasic.PCAN_USBBUS1)
If result <> TPCANStatus.PCAN_ERROR_QRCVEMPTY
    ' Process the received message
    MessageBox.Show("A message was received")
    ProcessMessage(msg)
Else
    ' An error occurred, get a text describing
    PCANBasic.GetErrorText(result, 0, strMsg)
    MessageBox.Show(strMsg.ToString())
    ' Here can be decided if the loop has to be
    ' status is bus-off
    HandleReadError(result)
End If
' Try to read a message from the receive queue of
' until the queue is empty
Loop While ((result And TPCANStatus.PCAN_ERROR_QRCVEMPTY) <> 0)

Pascal OO:

var
    result : TPCANStatus;
    strMsg: array [0..256] of Char;
    msg: TPCANMsgFD;
begin
    repeat
        // Check the receive queue for new message
        //
        result := PCANBasic.ReadFD(PCANBasic.PCAN_USBBUS1)
        If (result <> TPCANStatus.PCAN_ERROR_QRCVEMPTY) Then
            begin
                // Process the received message
                //
                MessageBox(0,'A message was received',
                ProcessMessage(msg);
            end
    end
else begin
  // An error occurred, get a text describing the error
  //
  TPCANBasic.GetErrorText(result, 0, strMsg);
  MessageBox(0, strMsg, 'Error', MB_OK);
  // Here can be decided if the loop has to be terminated (e.g. the bus status is bus-off)
  //
  HandleReadError(result);
end;
// Try to read a message from the receive queue of the PCAN-USB, Channel 1,
// until the queue is empty
//
until ((TPCANStatus(Integer(result) AND Integer(result)))

See Also

WriteFD
Using Events
Error Frames

Plain function Version: CAN_ReadFD
ReadFD(TPCANHandle, TPCANMsgFD, TPCANTimestampFD)

Reads a CAN message and its time stamp from the receive queue of a FD capable PCAN Channel.

Syntax

### Pascal OO

```pascal
class function ReadFD(
    Channel: TPCANHandle;
    var MessageBuffer: TPCANMsgFD;
    var TimestampBuffer: TPCANTimestampFD
): TPCANStatus; overload;
```

### C#

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_ReadFD")
public static extern TPCANStatus ReadFD(
    [MarshalAs(UnmanagedType.U1)]
    TPCANHandle Channel,
    out TPCANMsgFD MessageBuffer,
    out TPCANTimestampFD TimestampBuffer);
```

### C++/CLR

```cpp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_ReadFD")
static TPCANStatus ReadFD(
    [MarshalAs(UnmanagedType::U1)]
    TPCANHandle Channel,
    TPCANMsgFD %MessageBuffer,
    TPCANTimestampFD %TimestampBuffer);
```

### Visual Basic

```vbnet
<DllImport("PCANBasic.dll", EntryPoint:="CAN_ReadFD")
```
Public Shared Function ReadFD(
    <MarshalAs(UnmanagedType.U1)>
    ByVal Channel As TPCANHandle,  
    ByRef MessageBuffer As TPCANMsgFD,  
    ByRef TimestampBuffer As TPCANTimestampFD)
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a FD capable PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>MessageBuffer</td>
<td>A TPCANMsgFD buffer to store the CAN message.</td>
</tr>
<tr>
<td>TimestampBuffer</td>
<td>A TPCANTimestampFD buffer to get the reception time of the message.</td>
</tr>
</tbody>
</table>

Returns

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

- **PCAN_ERROR_IILLPARAMVAL:** Indicates that the parameters passed to the method are invalid. Check the value of the MessageBuffer; it should point to a TPCANMsgFD structure.
- **PCAN_ERROR_ILOOPERATION:** Indicates that the PCAN Channel passed to the method was not initialized.
using InitializeFD (plain function: CAN_InitializeFD).

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSWARNING:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-warning status.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSPASSIVE:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-passive status.</td>
</tr>
<tr>
<td>PCAN_ERRORBUSOFF:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-off status.</td>
</tr>
<tr>
<td>PCAN_ERROR_QRCVEMPTY:</td>
<td>Indicates that the receive queue of the Channel is empty.</td>
</tr>
</tbody>
</table>

**Remarks**

The use of Read and ReadFD are mutually exclusive. The PCAN Channel passed to this method must be initialized using InitializeFD (plain function: CAN_InitializeFD). Otherwise the error PCAN_ERROR_ILLOPERATION is returned.

If the time when the message was received is not needed, use the overloaded ReadFD method.

The ReadFD method returns received messages or status messages from the receive queue. It is important to call ReadFD repeatedly until the queue is empty. In case there are no more messages in queue,
the value PCAN_ERROR_QRCVEMPTY is returned. The error code
PCAN_ERROR_QRCVEMPTY is also returned if the reception of
messages is disabled. See Receive Status Parameter for more
information.

The receive queue can contain up to 32767 messages.

There are two possibilities for reading messages from the receive
queue of a Channel:

*Time-Triggered Reading*: Consists in periodically calls to the ReadFD
method. Typically, an application start a timer that every 50 or 100
milliseconds check for messages, calling the ReadFD method in a
loop until the value of PCAN_ERROR_QRCVEMTY or another error
condition is reached.

*Event-Triggered Reading*: Consists in reacting to a notification sent by
the PCAN driver to a registered application, when a message is
received and inserted in its receive queue. See Using Events to
obtain more information about reading with events.

**About bus errors / Status messages**

If a bus-off error occur, an application cannot use the channel to
communicate anymore, until the CAN controller is reset. With PCAN-
Basic it is not possible to reset the CAN controller through a function
directly. 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 BUSOFF, BUSWARNING, and
BUSPASSIVE, is to uninitialized and initialise again the channel used.
This causes a hardware reset.

The message type (see TPCANMessageType) of a CAN message
indicates if the message is a 11-bit, 29-bit, FD, RTR, Error, or Status
message. This value should be checked every time a message has
been read successfully.

If the bit PCAN_MESSAGE_ERRFRAME is set in the
TPCANMsg.MSGTYPE field, the message is an Error frame (see
If the bit `PCAN_MESSAGE_STATUS` is set in the `TPCANMsg.MSGTYPE` field, the message is a Status message. The ID and DLC fields do not contain valid data. The first 4 data bytes of the message contain the Error Code. The MSB of the Error Code is in data byte 0, the LSB is in data byte 3. If a status message was read the return value of `ReadFD` is also the error code.

**Examples:**

<table>
<thead>
<tr>
<th>Data0</th>
<th>Data1</th>
<th>Data2</th>
<th>Data3</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>02h</td>
<td>PCAN_ERROR_OVERRUN</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>08h</td>
<td>PCAN_ERROR_BUSWARNING</td>
</tr>
<tr>
<td>00h</td>
<td>04h</td>
<td>00h</td>
<td>00h</td>
<td>PCAN_ERROR_BUSPASSIVE</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>10h</td>
<td>PCAN_ERROR_BUSOFF</td>
</tr>
</tbody>
</table>

Example
The following example shows the use of method `ReadFD` on the channel PCAN_USBBUS1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: It is assumed that the channel was already initialized using the method `InitializeFD` and that the following code is an OnTimer event handler method.

**C#:**

```csharp
TPCANStatus result;
StringBuilder strMsg;
TPCANMsgFD msg;
TPCANTimestampFD time;

strMsg = new StringBuilder(256);

do {
    // Check the receive queue for new messages
    //
    result = PCANBasic.ReadFD(PCANBasic.PCAN_USBBUS1)
    if (result != TPCANStatus.PCAN_ERROR_QRCVEMPTY)
    {
        // Process the received message
        //
        MessageBox.Show("A message was received");
        ProcessMessage(msg);
    }
    else
    {
        // An error occurred, get a text describing
        //
        PCANBasic.GetErrorText(result, 0, strMsg);
        MessageBox.Show(strMsg.ToString());
        // Here can be decided if the loop has to
```
// status is bus-off
//
HandleReadError(result);
}
// Try to read a message from the receive queue of
// until the queue is empty
//
}while((result & TPCANStatus.PCAN_ERROR_QRCVEMPTY)

C++CLR:

TPCANStatus result;
StringBuilder^ strMsg;
TPCANMsgFD msg;
TPCANTimestampFD time;

strMsg = gcnew StringBuilder(256);

do
{
    // Check the receive queue for new messages
    //
    result = PCANBasic::ReadFD(PCANBasic::PCAN_USB
    if (result != TPCANStatus::PCAN_ERROR_QRCVEMPTY
    {
        // Process the received message
        //
        MessageBox::Show("A message was received")
        ProcessMessage(msg);
    }
    else
    {
        // An error occurred, get a text describin
        //
        PCANBasic::GetErrorText(result, 0, strMsg)
        MessageBox::Show(strMsg->ToString());
        // Here can be decided if the loop has to
        // status is bus-off)
HandleReadError(result);

// Try to read a message from the receive queue of
// until the queue is empty
//
}while((result & TPCANStatus::PCAN_ERROR_QRCVEMPTY)

Visual Basic:

Dim result As TPCANStatus
Dim strMsg As StringBuilder
Dim msg As TPCANMsgFD
Dim time As TPCANTimestampFD

strMsg = New StringBuilder(256)

Do
  ' Check the receive queue for new messages
  result = PCANBasic.ReadFD(PCANBasic.PCAN_USBBUS1)
  If result <> TPCANStatus.PCAN_ERROR_QRCVEMPTY Then
    MessageBox.Show("A message was received")
    ProcessMessage(msg)
  Else
    ' An error occurred, get a text describing
    ' the error
    PCANBasic.GetErrorText(result, 0, strMsg)
    MessageBox.Show(strMsg.ToString())
  End If
  ' Try to read a message from the receive queue of
  ' until the queue is empty
  Loop While ((result And TPCANStatus.PCAN_ERROR_QRCVEMPTY)
Pascal OO:

var
result : TPCANStatus;
strMsg: array [0..256] of Char;
msg: TPCANMsgFD;
time: TPCANTimestampFD;
begin
repeat
  // Check the receive queue for new message
  //
  result := TPCANBasic.ReadFD(TPCANBasic.PCAN_USB_BUS1);
  If (result <> PCAN_ERROR_QRCVEMPTY) Then
  begin
    // Process the received message
    //
    MessageBox(0,'A message was received',
                ProcessMessage(msg);
  end
  else
  begin
    // An error occurred, get a text describing the error and show it
    //
    TPCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox(0, strMsg, 'Error', MB_OK);
    // Here can be decided if the loop has to be terminated
    // status is bus-off
    //
    HandleReadError(result);
  end;
  // Try to read a message from the receive queue
  // until the queue is empty
  //
  until (TPCANStatus(Integer(result) AND Integer)
WriteFD

Using Events

Error Frames

Plain function Version: CAN_ReadFD
ReadFD(TPCANHandle)

Reads a CAN message and its time stamp from the receive queue of a FD capable PCAN Channel.

Syntax

```python
def ReadFD(
    self,
    Channel
)
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a FD capable PCAN Channel (see TPCANHandle).</td>
</tr>
</tbody>
</table>

Returns

The return value is a 3-touple. The order of the returned values is as follow:

[0]: The method's return value as a TPCANStatus code. PCAN_ERROR_OK is returned on success. The typical errors in case of failure are:

<p>| PCAN_ERROR_IILLPARAMVAL: | Indicates that the parameters passed to the method are invalid. Check the value of the MessageBuffer; it should point to a TPCANMsgFD structure. |</p>
<table>
<thead>
<tr>
<th>PCAN_ERROR_ILLOPERATION:</th>
<th>Indicates that the PCAN Channel passed to the method was not initialized using <code>InitializeFD</code> (plain function: <code>CAN_InititalizeFD</code>).</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSWARNING:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-warning status.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSPASSIVE:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-passive status.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSSOFF:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-off status.</td>
</tr>
<tr>
<td>PCAN_ERROR_QRCVEMPTY:</td>
<td>Indicates that the receive queue of the Channel is empty.</td>
</tr>
</tbody>
</table>

[1]: A `TPCMmsgFD` structure with the CAN message read.

[2]: A `TPCANTimestampFD` value with the time when a message was read.

Remarks

The use of `Read` and `ReadFD` are mutually exclusive. The PCAN Channel passed to this method **must be** initialized using `InitializeFD` (plain function: `CAN_InititalizeFD`). Otherwise the error
PCAN_ERROR_ILLOPERATION is returned.

The ReadFD method returns received messages or status messages from the receive queue. It is important to call ReadFD repeatedly until the queue is empty. In case there are no more messages in queue, the value PCAN_ERROR_QRCVEMPTY is returned. The error code PCAN_ERROR_QRCVEMPTY is also returned if the reception of messages is disabled. See Receive Status Parameter for more information.

The receive queue can contain up to 32767 messages.

There are two possibilities for reading messages from the receive queue of a Channel:

*Time-Triggered Reading*: Consists in periodically calls to the ReadFD method. Typically, an application start a timer that every 50 or 100 milliseconds check for messages, calling the ReadFD method in a loop until the value of PCAN_ERROR_QRCVEMTY or another error condition is reached.

*Event-Triggered Reading*: Consists in reacting to a notification sent by the PCAN driver to a registered application, when a message is received and inserted in its receive queue. See Using Events to obtain more information about reading with events.

**About bus errors / Status messages**

If a bus-off error occur, an application cannot use the channel to communicate anymore, until the CAN controller is reset. With PCAN-Basic it is not possible to reset the CAN controller through a function directly. 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 BUSOFF, BUSWARNING, and BUSPASSIVE, is to uninitialize and initialise again the channel used. This causes a hardware reset.

The message type (see TPCANMessageType) of a CAN message indicates if the message is a 11-bit, 29-bit, FD, RTR, Error, or Status
message. This value should be checked every time a message has been read successfully.

If the bit **PCAN_MESSAGE_ERRFRAME** is set in the **TPCANMsg.MSGTYPE** field, the message is an Error frame (see **Error Frames**).

If the bit **PCAN_MESSAGE_STATUS** is set in the **TPCANMsg.MSGTYPE** field, the message is a Status message. The ID and DLC fields do not contain valid data. The first 4 data bytes of the message contain the Error Code. The MSB of the Error Code is in data byte 0, the LSB is in data byte 3. If a status message was read the return value of **ReadFD** is also the error code.

Examples:

<table>
<thead>
<tr>
<th>Data0</th>
<th>Data1</th>
<th>Data2</th>
<th>Data3</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>02h</td>
<td>PCAN_ERROR_OVERRUN</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>08h</td>
<td>PCAN_ERROR_BUSWARNING</td>
</tr>
<tr>
<td>00h</td>
<td>04h</td>
<td>00h</td>
<td>00h</td>
<td>PCAN_ERROR_BUSPASSIVE</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>10h</td>
<td>PCAN_ERROR_BUSOFF</td>
</tr>
</tbody>
</table>
Python Notes

- Class-Method: Unlike the .NET Framework, under Python a variable has to be instantiated with an object of type PCANBasic in order to use the API functionality.
- Python's first argument convention: Under Python, 'self' is a parameter that is automatically included within the call of this method, within a PCANBasic object and hasn't to be indicated in a method call. This parameter represents the calling object itself.

Example

The following example shows the use of method ReadFD on the channel PCAN_USBBUS1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: It is assumed that the channel was already initialized using the method InitializeFD and that the following code is executed periodically.

Python:

```python
readResult = PCAN_ERROR_OK,
while (readResult[0] & PCAN_ERROR_QRCVEMPTY) != PCAN_ERROR_OK:
    # Check the receive queue for new messages
    readResult = objPCAN.ReadFD(PCAN_USBBUS1)
    if readResult[0] != PCAN_ERROR_QRCVEMPTY:
        # Process the received message
        #
        print "A message was received"
        ProcessMessage(result[1], result[2]) # Possible processing function
    else:
        # An error occurred, get a text describing
```
# result = objPCAN.GetErrorText(readResult[0])
print result[1]
HandleReadError(readResult[0])  # Possible

See Also

- WriteFD
- Using Events
- Error Frames

Plain function Version: CAN_ReadFD
Transmits a CAN message.

Syntax

**Pascal OO**

```pascal
class function Write(
    Channel: TPCANHandle;
    var MessageBuffer: TPCANMsg
): TPCANStatus;
```

**C#**

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_Write")]
public static extern TPCANStatus Write(
    [MarshalAs(UnmanagedType.U1)] TPCANHandle Channel,
    ref TPCANMsg MessageBuffer);
```

**C++ / CLR**

```cpp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_Write")]
static TPCANStatus Write(
    [MarshalAs(UnmanagedType::U1)] TPCANHandle Channel,
    TPCANMsg %MessageBuffer);
```

**Visual Basic**

```vbnet
<DllImport("PCANBasic.dll", EntryPoint:="CAN_Write")>
Public Shared Function Write(_
    <MarshalAs(UnmanagedType.U1)> _
    ByVal Channel As TPCANHandle, _
    ByVal MessageBuffer As TPCANMsg) As TPCANStatus
End Function
```
```python
def Write(
    self,
    Channel,
    MessageBuffer)
```

## Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>MessageBuffer</td>
<td>A TPCANMsg buffer containing the CAN message to be sent.</td>
</tr>
</tbody>
</table>

## Returns

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

<table>
<thead>
<tr>
<th>PCAN_ERROR_IILLPARAMVAL:</th>
<th>Indicates that the parameters passed to the method are invalid. Check the value of the MessageBuffer; it should point to a TPCANMsg structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td>PCAN_ERROR_QXMTFULL:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-off status.</td>
</tr>
<tr>
<td>PCAN_ERROR_QXMTFULL:</td>
<td>Indicates that the transmit</td>
</tr>
</tbody>
</table>
Remarks

If a bus-off error occurs, an application cannot use the channel to communicate anymore, until the CAN controller is reset. With PCAN-Basic it is not possible to reset the CAN controller through a method directly. 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 BUSOFF, BUSHEAVY, and BUSLIGTH, is to `uninitialize` and `initialize` again the channel used. This causes a hardware reset.

Python Notes

- Class-Method: Unlike the .NET Framework, under Python a variable has to be instantiated with an object of type `PCANBasic` in order to use the API functionality.
- Python's first argument convention: Under Python, `self` is a parameter that is automatically included within the call of this method, within a `PCANBasic` object and hasn't to be indicated in a method call. This parameter represents the calling object itself.

Example

The following example shows the use of the method Write on the channel PCAN_USBBUS1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

C#:

```csharp
TPCANStatus result;
StringBuilder strMsg;
```
TPCANMsg msg;

strMsg = new StringBuilder(256);

// A CAN message is created. The Data field (8 bytes)
// of the message must also be created
//
msg = new TPCANMsg();
msg.DATA = new Byte[8];

// A CAN message is configured
//
msg.ID = 0x100;
msg.MSGTYPE = TPCANMessageType.PCAN_MESSAGE_STANDARD;
msg.LEN = 3;
msg.DATA[0] = 1;
msg.DATA[1] = 2;
msg.DATA[2] = 3;

// The message is sent using the PCAN-USB Channel
//
result = PCANBasic.Write(PCANBasic.PCAN_USB_BUS1, ref if (result != TPCANStatus.PCAN_ERROR_OK)
{ // An error occurred, get a text describing the
// PCANBasic.GetErrorText(result, 0, strMsg);
MessageBox.Show(strMsg.ToString());
}
else
MessageBox.Show("Message sent successfully");

C++/CLR:

TPCANStatus result;
StringBuilder^ strMsg;
TPCANMsg^ msg;
strMsg = gcnew StringBuilder(256);

// A CAN message is created. The Data field (8 byte of the message must also be created
//
msg = gcnew TPCANMsg();
msg->DATA = gcnew array(Byte>(8);

// A CAN message is configured
//
msg->ID = 0x100;
msg->MSGTYPE = TPCANMessageType::PCAN_MESSAGE_STANDARD;
msg->LEN = 3;
msg->DATA[0] = 1;
msg->DATA[1] = 2;
msg->DATA[2] = 3;

// The message is sent using the PCAN-USB Channel
//
result = PCANBasic::Write(PCANBasic::PCAN_USBBUS1,
if (result != TPCANStatus::PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the
    //
    PCANBasic::GetErrorText(result, 0, strMsg);
    MessageBox::Show(strMsg->ToString());
}
else
    MessageBox::Show("Message sent successfully");

Visual Basic:

Dim result As TPCANStatus
Dim strMsg As StringBuilder
Dim msg As TPCANMsg

strMsg = New StringBuilder(256)
'A CAN message is created. The Data field (8 bytes) of the message must also be created'

```vbnet
msg = New TPCANMsg
msg.DATA = CType(Array.CreateInstance(GetType(Byte), 8), Byte)
```

'A CAN message is configured'

```vbnet
msg.ID = &H100
msg.MSGTYPE = TPCANMessageType.PCAN_MESSAGE_STANDARD
msg.LEN = 3
msg.DATA(0) = 1
msg.DATA(1) = 2
msg.DATA(2) = 3
```

'The message is sent using the PCAN-USB Channel 1'

```vbnet
result = PCANBasic.Write(PCANBasic.PCAN_USBBUS1, msg)
If result <> TPCANStatus.PCAN_ERROR_OK Then
    ' An error occurred, get a text describing the error
    PCANBasic.GetErrorText(result, 0, strMsg)
    MessageBox.Show(strMsg.ToString)
Else
    MessageBox.Show("Message sent successfully")
End If
```

Pascal OO:

```pascal
var
    result : TPCANStatus;
    strMsg: array [0..256] of Char;
    msg: TPCANMsg;
begin
    // A CAN message is configured
    msg.ID := $100;
    msg.MSGTYPE := PCAN_MESSAGE_STANDARD;
```
msg.LEN := 3;
msg.DATA[0] := 1;
msg.DATA[1] := 2;
msg.DATA[2] := 3;

// The message is sent using the PCAN-USB Channel
//
result := TPCANBasic.Write(TPCANBasic.PCAN_USB
If (result <> PCAN_ERROR_OK) Then
begin
  // An error occurred, get a text describing the error and show it
  TPCANBasic.GetErrorText(result, 0, strMsg)
  MessageBox(0, strMsg, 'Error', MB_OK);
end
else
  MessageBox(0, 'Message sent successfully',

Python:

# A CAN message is configured
#
msg = TPCANMsg()
msg.ID = 0x100
msg.MSGTYPE = PCAN_MESSAGE_STANDARD
msg.LEN = 3
msg.DATA[0] = 1
msg.DATA[1] = 2
msg.DATA[2] = 3

# The message is sent using the PCAN-USB Channel
#
result = objPCAN.Write(PCAN_USBBUS1, msg)
if result != PCAN_ERROR_OK:
  # An error occurred, get a text describing the error
  result = objPCAN.GetErrorText(result)
print result
else:
    print "Message sent successfully"

See Also

Read
SetValue

Plain function Version: CAN_Write

Copyright © 2017. PEAK-System Technik GmbH. All rights reserved.
Send feedback to this documentation
Transmits a CAN message using a FD capable PCAN Channel.

Syntax

**Pascal OO**

```pascal
class function WriteFD(
    Channel: TPCANHandle;
    var MessageBuffer: TPCANMsgFD
): TPCANStatus;
```

**C#**

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_WriteFD")
public static extern TPCANStatus WriteFD(
    [MarshalAs(UnmanagedType.U1)] TPCANHandle Channel,
    ref TPCANMsgFD MessageBuffer);
```

**C++ / CLR**

```cpp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_WriteFD")
static TPCANStatus WriteFD(
    [MarshalAs(\text{UnmanagedType}\::\text{U1})]
    TPCANHandle Channel,
    TPCANMsgFD %MessageBuffer);
```

**Visual Basic**

```vbnet
<DllImport("PCANBasic.dll", EntryPoint:="CAN_WriteFD")>
Public Shared Function WriteFD(_
    _
    ByVal Channel As TPCANHandle, _
    ByRef MessageBuffer As TPCANMsgFD) As TPCANStatus
End Function
```
```python
def WriteFD(
    self,
    Channel,
    MessageBuffer
)
```

### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a FD capable PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>MessageBuffer</td>
<td>A TPCANMsgFD buffer containing the CAN message to be sent.</td>
</tr>
</tbody>
</table>

### Returns

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

- **PCAN_ERROR_IILLPARAMVAL**: Indicates that the parameters passed to the method are invalid. Check the value of the MessageBuffer; it should point to a TPCANMsgFD structure.
- **PCAN_ERROR_ILLOPERATION**: Indicates that the PCAN Channel passed to the method was not initialized using InitializeFD (plain function: CAN_InitializeFD).
- **PCAN_ERROR_INITIALIZE**: Indicates that the given
PCAN channel was not found in the list of initialized channels of the calling application.

**PCAN_ERROR_BUSOFF:** Indicates a bus error within the given PCAN Channel. The hardware is in bus-off status.

**PCAN_ERROR_QXMTFULL:** Indicates that the transmit queue of the Channel is full.

**Remarks**

The use of `Write` and `WriteFD` are mutually exclusive. The PCAN Channel passed to this method **must be** initialized using `InitializeFD` ([plain function: CAN_InitializeFD](#)). Otherwise the error `PCAN_ERROR_ILLOPERATION` is returned.

If a bus-off error occur, an application cannot use the channel to communicate anymore, until the CAN controller is reset. With PCAN-Basic it is not possible to reset the CAN controller through a method directly. 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 BUSOFF, BUSWARNING, and BUSPASSIVE, is to `uninitialize` and `initialise` again the channel used. This causes a hardware reset.

**Python Notes**

- **Class-Method:** Unlike the .NET Framework, under Python a variable has to be instantiated with an object of type `PCANBasic` in order to use the API functionality.
- Python's first argument convention: Under Python, ‘self’ is a parameter that is automatically included within the call of this method, within a `PCANBasic` object and hasn't to be indicated in
a method call. This parameter represents the calling object itself.

Example

The following example shows the use of the method WriteFD on the channel PCAN_USBUSB1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

C#:

```csharp
TPCANStatus result;
StringBuilder strMsg;
TPCANMsgFD msg;

strMsg = new StringBuilder(256);

// A CAN message is created. The Data field (64 by
// of the message must also be created
//
msg = new TPCANMsgFD();
msg.DATA = new Byte[64];

// A CAN message is configured
//
msg.ID = 0x100;
msg.MSGTYPE = TPCANMessageType.PCAN_MESSAGE_STANDARD

// DLC 9 means 12 data bytes
//
msg.DLC = 9;
for(byte i=0; i < 12; i++)
    msg.DATA[i] = i;

// The message is sent using the PCAN-USB Channel
```
result = PCANBasic.WriteFD(PCANBasic.PCAN_USB_BUS1, result != TPCANStatus.PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    PCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox.Show(strMsg.ToString());
} else
    MessageBox.Show("Message sent successfully");

C++/CLR:

TPCANStatus result;
StringBuilder^ strMsg;
TPCANMsgFD^ msg;

strMsg = gcnew StringBuilder(256);

// A CAN message is created. The Data field (64 by
// of the message must also be created
//
msg = gcnew TPCANMsgFD();
msg->DATA = gcnew array<Byte>(64);

// A CAN message is configured
//
msg->ID = 0x100;
msg->MSGTYPE = TPCANMessageType::PCAN_MESSAGE_STANDARD

// DLC 9 means 12 data bytes
//
msg->DLC = 9;
for(Byte i=0; i < 12; i++)
    msg->DATA[i] = i;

// The message is sent using the PCAN-USB Channel
```csharp
//
result = PCANBasic::WriteFD(PCANBasic::PCAN_USBBUS1
if (result != TPCANStatus::PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    //
    PCANBasic::GetErrorText(result, 0, strMsg);
    MessageBox::Show(strMsg->ToString());
}
else
    MessageBox::Show("Message sent successfully");
```

### Visual Basic:

```vbnet
Dim result As TPCANStatus
Dim strMsg As StringBuilder
Dim msg As TPCANMsgFD

strMsg = New StringBuilder(256)

' A CAN message is created. The Data field (64 bytes) of the message must also be created
'
msg = New TPCANMsgFD
msg.DATA = CType(Array.CreateInstance(GetType(Byte), 64)

' A CAN message is configured
'
msg.ID = &H100
msg.MSGTYPE = TPCANMessageType.PCAN_MESSAGE_STANDARD

' DLC 9 means 12 data bytes
'
msg.DLC = 9
For i As Byte = 0 To 11
    msg.DATA(i) = i
Next
```
'The message is sent using the PCAN-USB Channel 1
'
result = PCANBasic.WriteFD(PCANBasic.PCAN_USBBUS1, msg)
If result <> TPCANStatus.PCAN_ERROR_OK Then
    ' An error occurred, get a text describing the error
    PCANBasic.GetErrorText(result, 0, strMsg)
    MessageBox.Show(strMsg.ToString)
Else
    MessageBox.Show("Message sent successfully")
End If

Pascal OO:

var
    result : TPCANStatus;
    strMsg: array [0..256] of Char;
    msg: TPCANMsgFD;
    I: Integer;
begin
    // A CAN message is configured
    //
    msg.ID := $100;
    msg.MSGTYPE := TPCANMessageType(Byte(PCAN_MESSAGE_STANDARD))
    // DLC 9 means 12 data bytes
    //
    msg.DLC := 9;
    for I:=0 To 11 do
        msg.DATA[I] := I;

    // The message is sent using the PCAN-USB Channel
    //
    result := TPCANBasic.WriteFD(TPCANBasic.PCAN_USBBUS1, msg)
    If (result <> PCAN_ERROR_OK) Then
        begin
            // An error occurred, get a text describing the error
        end
    //
TPCANBasic.GetErrorText(result, 0, strMsg);
MessageBox(0, strMsg, 'Error',MB_OK);
end
else
MessageBox(0,'Message sent successfully',"

Python:

```python
# A CAN message is configured
#
msg = TPCANMsgFD()
msg.ID = 0x100
msg.MSGTYPE = PCAN_MESSAGE_STANDARD.value | PCAN_MESSAGE_FD

# DLC 9 means 12 data bytes
#
msg.DLC = 9
for i in range(12):
    msg.DATA[i] = i

# The message is sent using the PCAN-USB Channel
#
result = objPCAN.WriteFD(PCAN_USBBUS1,msg)
if result != PCAN_ERROR_OK:
    # An error occurred, get a text describing the
    #
    result = objPCAN.GetErrorText(result)
    print result
else:
    print "Message sent successfully"
```

See Also

- ReadFD
- SetValue
Plain function Version: **CAN_WriteFD**
GetValue

Retrieves information from a PCAN Channel.

### Overloads

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GetValue(TPCANHandle, TPCANParameter, String, UInt32)</code></td>
<td>Retrieves information from a PCAN Channel in text form.</td>
</tr>
<tr>
<td><code>GetValue(TPCANHandle, TPCANParameter, UInt32, UInt32)</code></td>
<td>Retrieves information from a PCAN Channel in numeric form (32-Bit).</td>
</tr>
<tr>
<td><code>GetValue(TPCANHandle, TPCANParameter, UInt64, UInt32)</code></td>
<td>Retrieves information from a PCAN Channel in numeric form (64-Bit).</td>
</tr>
<tr>
<td><code>GetValue(TPCANHandle, TPCANParameter)</code></td>
<td>Retrieves information from a PCAN Channel.</td>
</tr>
</tbody>
</table>
**GetValue(TPCANHandle, TPCANParameter, String, UInt32)**

Retrieves information from a PCAN Channel in text form.

**Syntax**

<table>
<thead>
<tr>
<th>Pascal OO</th>
</tr>
</thead>
<tbody>
<tr>
<td>class function GetValue(</td>
</tr>
<tr>
<td>Channel: TPCANHandle;</td>
</tr>
<tr>
<td>Parameter: TPCANParameter;</td>
</tr>
<tr>
<td>StringBuffer: PChar;</td>
</tr>
<tr>
<td>BufferLength: LongWord</td>
</tr>
<tr>
<td>): TPCANStatus; overload;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static extern TPCANStatus GetValue(</td>
</tr>
<tr>
<td>[MarshalAs(UnmanagedType.U1)] TPCANHandle Channel,</td>
</tr>
<tr>
<td>[MarshalAs(UnmanagedType.U1)] TPCANParameter Parameter,</td>
</tr>
<tr>
<td>StringBuilder StringBuffer,</td>
</tr>
<tr>
<td>UInt32 BufferLength);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C++ / CLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>[DllImport(&quot;PCANBasic.dll&quot;, EntryPoint = &quot;CAN_GET_VALUE&quot;),</td>
</tr>
<tr>
<td>MarshalAs(UnmanagedType::U1)] TPCANHandle Channel,</td>
</tr>
<tr>
<td>MarshalAs(UnmanagedType::U1)] TPCANParameter Parameter,</td>
</tr>
<tr>
<td>StringBuilder^ StringBuffer,</td>
</tr>
<tr>
<td>UInt32 BufferLength);</td>
</tr>
</tbody>
</table>
# Visual Basic

```vbnet
<DllImport("PCANBasic.dll", EntryPoint:="CAN_GetValue")>
Public Shared Function GetValue( 
    <MarshalAs(UnmanagedType.U1)> _ ByVal Channel As TPCANHandle, _ 
    <MarshalAs(UnmanagedType.U1)> _ ByVal Parameter As TPCANParameter, _ 
    ByVal StringBuffer As StringBuilder, _ 
    ByVal BufferLength As UInt32) As TPCANStatus
End Function
```

## Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the information to be retrieved (see TPCANParameter).</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 TPCANStatus code. PCAN_ERROR_OK is returned on success. The typical errors in case of failure are:

- **PCAN_ERROR_I LLPARMVAL**: Indicates that the
parameters passed to the method are invalid. Check the parameter 'StringBuffer'; it should point to a valid null-terminated string buffer.

<table>
<thead>
<tr>
<th><strong>PCAN_ERROR_INITIALIZE:</strong></th>
<th>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PCAN_ERROR_ILLPARAMTYPE:</strong></td>
<td>Indicates that the requested information is not available for the given PCAN Channel. Check the value of 'Parameter'; some values are not available for all PCAN-Channels or cannot be read.</td>
</tr>
</tbody>
</table>

**Remarks**

Use the method **GetValue** to get information about PCAN environment as parameters like the Message Filter and values like the availability of a PCAN-Channel. Take in account that not all parameters are supported for all PCAN-Channels. The access's type of the parameters can also be different.

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

**Example**

The following example shows the use of the method **GetValue** on the channel PCAN_USBBUS1 to get the PCAN-Channel version text. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.
Note: It is assumed that the channel was already initialized

**InitializeFD.**

**C#:**

```csharp
TPCANStatus result;
StringBuilder strMsg;

strMsg = new StringBuilder(256);

// The version of the PCAN-USB Channel 1 is asked.
//
result = PCANBasic.GetValue(PCANBasic.PCAN_USBBUS1)
if (result != TPCANStatus.PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error and show it
    //
    PCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox.Show(strMsg.ToString());
}
else
{
    // Show the version message
    //
    MessageBox.Show(strMsg.ToString());
}
```

**C++/CLR:**

```c++
TPCANStatus result;
StringBuilder^ strMsg;

strMsg = gcnew StringBuilder(256);

// The version of the PCAN-USB Channel 1 is asked.
//
result = PCANBasic::GetValue(PCANBasic::PCAN_USBBUS1)
if (result != TPCANStatus::PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error and show it
    //
    PCANBasic::GetErrorText(result, 0, strMsg);
    MessageBox.Show(strMsg->ToString());
}
else
{
    // Show the version message
    //
    MessageBox.Show(strMsg->ToString());
}
PCANBasic::GetErrorText(result, 0, strMsg);
MessageBox::Show(strMsg->ToString());
}
else

// Show the version message
//
MessageBox::Show(strMsg->ToString());

Visual Basic:

Dim result As TPCANStatus
Dim strMsg As StringBuilder

strMsg = New StringBuilder(256)

' The version of the PCAN-USB Channel 1 is asked.
'
result = PCANBasic.GetValue(PCANBasic.PCAN_USB_BUS1)
If result <> TPCANStatus.PCAN_ERROR_OK Then

' An error occurred, get a text describing the
'
PCANBasic.GetErrorText(result, 0, strMsg)
MessageBox.Show(strMsg.ToString())

Else

' Show the version message
'
MessageBox.Show(strMsg.ToString())
End If

Pascal OO:

var
result: TPCANStatus;
strMsg: array [0..256] of Char;

begin

// The version of the PCAN-USB Channel 1 is asked.
//
result := TPCANBasic.GetValue(TPCANBasic.PCAN_USBBUS1)
If (result <> PCAN_ERROR_OK) Then
begin
  // An error occurred, get a text describing
  // TPCANBasic.GetIntErrorMsg(result, 0, strMsg)
  MessageBox(0, strMsg, 'Error', MB_OK);
end
else
  // Show the version message
  // MessageBox(0, strMsg, 'Success', MB_OK);

See Also

SetValue
TPCANParameter
Parameter Value Definitions

Plain function Version: CAN_GetValue
**GetValue(TPCANHandle, TPCANParameter, UInt32, UInt32)**

Retrieves information from a PCAN Channel in numeric form.

**Syntax**

<table>
<thead>
<tr>
<th>Pascal OO</th>
</tr>
</thead>
</table>
| ```pascal
  class function GetValue(
    Channel: TPCANHandle;
    Parameter: TPCANParameter;
    NumericBuffer: PLongWord;
    BufferLength: LongWord
  ): TPCANStatus; overload;
``` |

<table>
<thead>
<tr>
<th>C#</th>
</tr>
</thead>
</table>
| ```csharp
  public static extern TPCANStatus GetValue(
    [MarshalAs(UnmanagedType.U1)] TPCANHandle Channel,
    [MarshalAs(UnmanagedType.U1)] TPCANParameter Parameter,
    out UInt32 NumericBuffer,
    UInt32 BufferLength);
``` |

<table>
<thead>
<tr>
<th>C++ / CLR</th>
</tr>
</thead>
</table>
| ```cpp
  [DllImport("PCANBasic.dll", EntryPoint = "CANGetValue")]
  static TPCANStatus GetValue(
    [MarshalAs(UnmanagedType::U1)] TPCANHandle Channel,
    [MarshalAs(UnmanagedType::U1)] TPCANParameter Parameter,
    UInt32 %NumericBuffer,
``` |
Visual Basic

<DllImport("PCANBasic.dll", EntryPoint:="CAN_GetValue")>
Public Shared Function GetValue( _
   <MarshalAs(UnmanagedType.U1)>
   ByVal Channel As TPCANHandle, _
   <MarshalAs(UnmanagedType.U1)>
   ByVal Parameter As TPCANParameter, _
   ByRef NumericBuffer As UInt32, _
   ByVal BufferLength As UInt32) As TPCANStatus
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the information to be retrieved (see TPCANParameter).</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 TPCANStatus code. PCAN_ERROR_OK is returned on success. The typical errors in case of failure are:

PCAN_ERROR_IILLPARAMVAL: Indicates that the
<table>
<thead>
<tr>
<th>Parameters passed to the method are invalid. Check the parameter 'NumericBuffer'; it should point to an integer buffer.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PCAN_ERROR_INITIALIZE:</strong> Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td><strong>PCAN_ERROR_ILLPARAMTYPE:</strong> Indicates that the requested information is not available for the given PCAN Channel. Check the value of 'Parameter'; some values are not available for all PCAN-Channels or cannot be read.</td>
</tr>
</tbody>
</table>

## Remarks

Use the method [GetValue](#) to get information about PCAN environment as parameters like the Message Filter and values like the availability of a PCAN-Channel. Take in account that not all parameters are supported for all PCAN-Channels. The access's type of the parameters can also be different.

More information about the parameters and values that can be read can be found in [Parameter Value Definitions](#).

## Example

The following example shows the use of the method [GetValue](#) on the channel PCAN_USBBUS1 to check if the Message Filter is fully opened. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.
Note: It is assumed that the channel was already initialized.

C#:

```csharp
TPCANStatus result;
StringBuilder strMsg;
UInt32 iBuffer;

strMsg = new StringBuilder(256);

// The status of the message filter of the PCAN-USB
// result = PCANBasic.GetValue(PCANBasic.PCAN_USBBUS1
if (result != TPCANStatus.PCAN_ERROR_OK)
{
    // An error occurred, get a text describing th
    //
    PCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox.Show(strMsg.ToString());
}
else
// A text is shown giving information about th
//
switch(iBuffer)
{
    case PCANBasic.PCAN_FILTER_OPEN:
        MessageBox.Show("The message filter fo break;
    case PCANBasic.PCAN_FILTER_CLOSE:
        MessageBox.Show("The message filter fo break;
    case PCANBasic.PCAN_FILTER_CUSTOM:
        MessageBox.Show("The message filter fo break;
}
```

C++/CLR:

```csharp
TPCANStatus result;
```
StringBuilder^ strMsg;
UInt32 iBuffer;

strMsg = gcnew StringBuilder(256);

// The status of the message filter of the PCAN-USB
//
result = PCANBasic::GetValue(PCANBasic::PCAN_USBBUS1
if (result != TPCANStatus::PCAN_ERROR_OK)
{
  // An error occurred, get a text describing th
  //
  PCANBasic::GetErrorText(result, 0, strMsg);
  MessageBox::Show(strMsg->ToString());
}
else
// A text is shown giving information about th
//
switch(iBuffer)
{
  case PCANBasic::PCAN_FILTER_OPEN:
    MessageBox::Show("The message filter f
  break;
  case PCANBasic::PCAN_FILTER_CLOSE:
    MessageBox::Show("The message filter f
  break;
  case PCANBasic::PCAN_FILTER_CUSTOM:
    MessageBox::Show("The message filter f
  break;

Visual Basic:

Dim result As TPCANStatus
Dim strMsg As StringBuilder
Dim iBuffer As UInt32

strMsg = New StringBuilder(256)
The status of the message filter of the PCAN-USB Channel 1 is asked.

result = PCANBasic.GetValue(PCANBasic.PCAN_USBBUS1)
If result <> TPCANStatus.PCAN_ERROR_OK Then
    ' An error occurred, get a text describing the error
    PCANBasic.GetErrorText(result, 0, strMsg)
    MessageBox.Show(strMsg.ToString())
Else
    ' A text is shown giving information about the current status
    Select Case iBuffer
        Case PCANBasic.PCAN_FILTER_OPEN
            MessageBox.Show("The message filter for the PCAN-USB, channel 1, is completely opened.
        Case PCANBasic.PCAN_FILTER_CLOSE
            MessageBox.Show("The message filter for the PCAN-USB, channel 1, is closed.
        Case PCANBasic.PCAN_FILTER_CUSTOM
            MessageBox.Show("The message filter for the PCAN-USB, channel 1, is custom configured.
    End Select
End If

Pascal OO:

var
    result : TPCANStatus;
    strMsg: array [0..256] of Char;
    iBuffer: LongWord;
begin
    // The status of the message filter of the PCAN-USB Channel 1 is asked
    result := PCANBasic.GetValue(TPCANBasic.PCAN_USBBUS1)
    If (result <> PCAN_ERROR_OK) Then
        begin
            // An error occurred, get a text describing the error
            //
            PCANBasic.GetErrorText(result, 0, strMsg)
            MessageBox(0, strMsg, 'Error', MB_OK);
end
else
begin

// A text is shown giving information about the current status

//
if iBuffer = TPCANBasic.PCAN_FILTER_OPEN then
  MessageBox(0, 'The message filter for the PCAN-USB, channel 1, is completely opened.'
if iBuffer = TPCANBasic.PCAN_FILTER_CLOSE
  MessageBox(0, 'The message filter for the PCAN-USB, channel 1, is closed.'
if iBuffer = TPCANBasic.PCAN_FILTER_CUSTOM
  MessageBox(0, 'The message filter for the PCAN-USB, channel 1, is custom configured.'
end;

See Also

SetValue
TPCANParameter
Parameter Value Definitions

Plain function Version: CANGetValue

Copyright © 2017. PEAK-System Technik GmbH. All rights reserved.
Send feedback to this documentation
GetValue(TPCANHandle, TPCANParameter, UInt64, UInt32)

Retrieves information from a PCAN Channel in numeric form.

#### Pascal OO

```pascal
class function GetValue(
    Channel: TPCANHandle;
    Parameter: TPCANParameter;
    NumericBuffer: PUInt64;
    BufferLength: LongWord
): TPCANStatus; overload;
```

#### C#

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_GetValue")
public static extern TPCANStatus GetValue(
    [MarshalAs(UnmanagedType.U1)] TPCANHandle Channel,
    [MarshalAs(UnmanagedType.U1)] TPCANParameter Parameter,
    out UInt64 NumericBuffer,
    UInt32 BufferLength);
```

#### C++ / CLR

```cpp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_GetValue")
static TPCANStatus GetValue(
    [MarshalAs(UnmanagedType::U1)] TPCANHandle Channel,
    [MarshalAs(UnmanagedType::U1)] TPCANParameter Parameter,
    UInt64 %NumericBuffer,
    UInt32 BufferLength);
```

#### Visual Basic

```vb
Option Explicit

Private Declare Function GetValue Lib "PCANBasic.dll" (ByVal Channel As Long, ByVal Parameter As Long, ByRef NumericBuffer As Long, ByVal BufferLength As Long) As Long
```
<DllImport("PCANBasic.dll", EntryPoint:="CAN_Ge"
Public Shared Function GetValue( _
  <MarshalAs(UnmanagedType.U1)> _
  ByVal Channel As TPCANHandle, _
  <MarshalAs(UnmanagedType.U1)> _
  ByVal Parameter As TPCANParameter, _
  ByRef NumericBuffer As UInt64, _
  ByVal BufferLength As UInt32) As TPCANStatus
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the information to be retrieved (see TPCANParameter).</td>
</tr>
<tr>
<td>NumericBuffer</td>
<td>The buffer to return the required 64-bit 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 TPCANStatus code. PCAN_ERROR_OK is returned on success. The typical errors in case of failure are:

| PCAN_ERROR_ILLPARAMVAL: | Indicates that the parameters passed to the method are invalid. Check the parameter |
'NumericBuffer'; it should point to a 64-bit integer buffer.

**PCAN_ERROR_INITIALIZE:** Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.

**PCAN_ERROR_ILLPARAMTYPE:** Indicates that the requested information is not available for the given PCAN Channel. Check the value of 'Parameter'; some values are not available for all PCAN-Channels or cannot be read.

 Remarks

Use the method **GetValue** to get information about PCAN environment as parameters like an Acceptance Filter. Take in account that not all parameters are supported for all PCAN-Channels. The access's type of the parameters can also be different.

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

 Example

The following example shows the use of the method **GetValue** on the channel PCAN_USBBUS1 to retrieve the configured message filter as 11-bit acceptance code and mask. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

```csharp
TPCANStatus result;
StringBuilder strMsg;
UInt64 i64Buffer;
UInt32 iCode, iMask;

strMsg = new StringBuilder(256);

// The 11-bit acceptance filter of the PCAN-USB Channel 1 is asked
result = PCANBasic.GetValue(PCANBasic.PCAN_USBBUS1);
if (result != TPCANStatus.PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    PCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox.Show(strMsg.ToString());
}
else
{
    // Code and mask are extracted from the 64-bit numeric value
    iCode = Convert.ToUInt32((i64Buffer >> 32);
    iMask = Convert.ToUInt32(i64Buffer & UInt32.MaxValue);

    // A text is shown giving information about the configured 11-bit acceptance filter
    MessageBox.Show(string.Format("Configured 11-bit acceptance filter:
    Code: {0:X8} | Mask: {1:X8}", iCode, iMask));
}
```

C++/CLR:

```cpp
TPCANStatus result;
StringBuilder^ strMsg;
UI64 i64Buffer;
UI32 iCode, iMask;
```
strMsg = gcnew StringBuilder(256);

// The 11-bit acceptance filter of the PCAN-USB Channel 1 is asked
result = PCANBasic::GetValue(PCANBasic::PCAN_USBBUS1);
if (result != TPCANStatus::PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    PCANBasic::GetErrorText(result, 0, strMsg);
    MessageBox::Show(strMsg->ToString());
}
else
{
    // Code and mask are extracted from the 64-bit numeric value
    iCode = Convert::ToUInt32((i64Buffer >> 32);
    iMask = Convert::ToUInt32(i64Buffer & UInt32::MaxValue);

    // A text is shown giving information about the filter
    MessageBox::Show(String::Format("Configured 11-bit acceptance filter: Code: {0:X8} | Mask: {1:X8}"), iCode, iMask);
}

Visual Basic:

Dim result As TPCANStatus
Dim strMsg As StringBuilder
Dim i64Buffer As UInt64
Dim iCode As UInt32
Dim iMask As UInt32

strMsg = New StringBuilder(256)

' The 11-bit acceptance filter of the PCAN-USB Channel 1 is asked
result = PCANBasic.GetValue(PCANBasic.PCAN_USBBUS1)
If result <> TPCANStatus.PCAN_ERROR_OK Then
An error occurred, get a text describing the error and show it with PCANBasic.

```pascal
Pascal OO:
var
  result : TPCANStatus;
  strMsg: array [0..256] of Char;
  i64Buffer: UInt64;
  iCode, iMask: LongWord;
begin
  // The 11-bit acceptance filter of the PCAN-US channel is asked.
  result := TPCANBasic.GetValue(TPCANBasic.PCAN_USB1);
  If (result <> PCAN_ERROR_OK) Then
  begin
    // An error occurred, get a text describing the error and show it.
    TPCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox(0, strMsg, 'Error', MB_OK);
  end
  else
  begin
    // Code and mask are extracted from the 64-bit numeric value.
    iCode := i64Buffer shr 32;
    // A text is shown giving information about the 11-bit mask and code of the acceptance filter.
    MessageBox.Show(String.Format("Configured 11-bit acceptance filter: Code: {0:X8} | Mask: {1:X8}"),
    iCode, iMask);
  end
```

```csharp
else
{    // Code and mask are extracted from the 64-bit numeric value.
    iCode = Convert.ToUInt32(i64Buffer >> 32);
    iMask = Convert.ToUInt32(i64Buffer And UInt32.MaxValue);

    // A text is shown giving information about the 11-bit mask and code of the acceptance filter.
    MessageBox.Show(String.Format("Configured 11-bit acceptance filter: Code: \{\0:X8\} | Mask: \{\1:X8\}"),
    iCode, iMask);
```
iMask := (i64Buffer And $FFFFFFFE);

// A text is shown giving information about
//
MessageBox(0, PChar(Format('Configured 11-bit acceptance filter: Code: %.8X | Mask: %.8X')));
end;

See Also

SetValue
TPCANParameter
Parameter Value Definitions

Plain function Version: CAN_GetValue
GetValue(TPCANHandle, TPCANParameter)

Retrieves information from a PCAN Channel.

Syntax

```python
def GetValue(self, Channel, Parameter)
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the information to be retrieved (see TPCANParameter).</td>
</tr>
</tbody>
</table>

Returns

The return value is a 2-touple. The order of the returned values is as follow:

[0]: The method's return value as a TPCANStatus code. PCAN_ERROR_OK is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>PCAN_ERROR_ILLPARAMVAL:</th>
<th>Indicates that the parameters passed to the method are invalid. Check the parameter</th>
</tr>
</thead>
</table>
'NumericBuffer'; it should point to an integer buffer.

<table>
<thead>
<tr>
<th>PCAN_ERROR_INITIALIZE:</th>
<th>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_ILLPARAMTYPE:</td>
<td>Indicates that the requested information is not available for the given PCAN Channel. Check the value of 'Parameter'; some values are not available for all PCAN-Channels or cannot be read.</td>
</tr>
</tbody>
</table>

[1]: The requested parameter value (the type of the value depends on the **TPCANParameter** requested).

**Remarks**

Use the method **GetValue** to get information about PCAN environment as parameters like the Message Filter and values like the availability of a PCAN-Channel. Take in account that not all parameters are supported for all PCAN-Channels. The access's type of the parameters can also be different.

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

**Python Notes**

- Class-Method: Unlike the .NET Framework, under Python a variable has to be instantiated with an object of type **PCANBasic** in order to use the API functionality.
- Python's first argument convention: Under Python, 'self' is a parameter that is automatically included within the call of this method, within a **PCANBasic** object and hasn't to be indicated in
a method call. This parameter represents the calling object itself.

**Example**

The following example shows the use of the method `GetValue` on the channel PCAN_USBBUS1 to check if the Message Filter is fully opened. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

**Python:**

```python
# The status of the message filter of the PCAN-USB
result = objPCAN.GetValue(PCAN_USBBUS1, PCAN_MESSAGE_FILTER)
if result[0] != PCAN_ERROR_OK:
    # An error occurred, get a text describing the error
    result = objPCAN.GetErrorText(result)
    print result
else:
    # A text is shown giving information about the current status
    if result[1] == PCAN_FILTER_OPEN:
        print "The message filter for the PCAN-USB is fully opened.
    elif result[1] == PCAN_FILTER_CLOSE:
        print "The message filter for the PCAN-USB is closed.
    elif result[1] == PCAN_FILTER_CUSTOM:
        print "The message filter for the PCAN-USB is custom configured."
```

**See Also**

- `SetValue`
- `TPCANParameter`
- `Parameter Value Definitions`
Plain function Version: CAN_GetValue
Sets a configuration or information value within a PCAN Channel.

**Overloads**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SetValue(TPCANHandle, TPCANParameter, String, UInt32)</code></td>
<td>Sets a configuration or information string value within a PCAN Channel.</td>
</tr>
<tr>
<td><code>SetValue(TPCANHandle, TPCANParameter, UInt32, UInt32)</code></td>
<td>Sets a configuration or information numeric value within a PCAN Channel (32-Bit).</td>
</tr>
<tr>
<td><code>SetValue(TPCANHandle, TPCANParameter, UInt64, UInt32)</code></td>
<td>Sets a configuration or information numeric value within a PCAN Channel (64-Bit).</td>
</tr>
<tr>
<td><code>SetValue(TPCANHandle, TPCANParameter, Object)</code></td>
<td>Sets a configuration or information value within a PCAN Channel.</td>
</tr>
</tbody>
</table>
**SetValue(TPCANHandle, TPCANParameter, String, UInt32)**

Sets a configuration or information string value within a PCAN Channel.

**Syntax**

**Pascal OO**

```pascal
class function SetValue(
    Channel: TPCANHandle;
    Parameter: TPCANParameter;
    StringBuffer: PChar;
    BufferLength: LongWord
): TPCANStatus; overload;
```

**C#**

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CANSetValue")]
public static extern TPCANStatus SetValue(
    [MarshalAs(UnmanagedType.U1)]
    TPCANHandle Channel,
    [MarshalAs(UnmanagedType.U1)]
    TPCANParameter Parameter,
    [MarshalAs(UnmanagedType.LPStr, SizeParamIndex=3)]
    string StringBuffer,
    UInt32 BufferLength);
```

**C++/CLR**

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CANSetValue")]
static TPCANStatus SetValue(
    [MarshalAs(UnmanagedType::U1)]
    TPCANHandle Channel,
    [MarshalAs(UnmanagedType::U1)]
    TPCANParameter Parameter,
    [MarshalAs(UnmanagedType::LPStr, SizeParamIndex=3)]
    string StringBuffer,
    UInt32 BufferLength);
```
```csharp
TPCANParameter Parameter,
MarshalAs(UnmanagedType::LPStr,SizeParamIndex=3)
String^ StringBuffer,
UInt32 BufferLength);

Visual Basic

<DllImport("PCANBasic.dll", EntryPoint:="CAN_SetValue"
Public Shared Function SetValue(_
    ByVal Channel As TPCANHandle, _
    ByVal Parameter As TPCANParameter, _
    ByVal StringBuffer As String, _
    ByVal BufferLength As UInt32) As TPCANStatus
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the value to be set (see TPCANParameter).</td>
</tr>
<tr>
<td>StringBuffer</td>
<td>The buffer containing the string 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 TPCANStatus code. PCAN_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>PCAN_ERROR_ILLPARAMVAL</td>
<td>Indicates that the parameters passed to the method are invalid. Check the parameter 'StringBuffer'; it should point to a valid null-terminated string buffer.</td>
</tr>
<tr>
<td>PCAN_ERROR_CAUTION</td>
<td>The configuration of a parameter failed due to a no more existing channel. The parameter has been reset on all existing channels.</td>
</tr>
<tr>
<td>PCAN_ERROR_INITIALIZE</td>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLPARAMTYPE</td>
<td>Indicates that the requested information is not available for the given PCAN Channel. Check the value of 'Parameter'; some values are not available for all PCAN-Channels or cannot be set.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLOPERATION</td>
<td>An underlying process that is generated by a call to this method with the current parameters, is temporarily not allowed. The configuration in relation to the used TPCANParameter must be checked.</td>
</tr>
</tbody>
</table>
Remarks

Use the method `SetValue` to set configuration information or environment values of a PCAN Channel as parameters like the Message Filter and values like a custom entry in the log file of PCAN-Basic. Take in account that not all parameters are supported for all PCAN-Channels. The access's type of the parameters can also be different.

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 PCAN_NONEBUS to set (and activate) the path for the log file of a PCAN-Basic's debug session. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: It is not needed to have an initialized PCAN channel for using the Log functionality.

C#:

```csharp
TPCANStatus result;
StringBuilder strMsg;
string strBuffer;

strMsg = new StringBuilder(256);

// The path for the Log file is set.
// Note that this parameter is set using the // default Channel (PCAN_NONEBUS)
//
strBuffer = "C:\Users\Admin\Desktop";
result = PCANBasic.SetValue(PCANBasic.PCAN_NONEBUS
if (result != TPCANStatus.PCAN_ERROR_OK)
C++/CLR:

```c++
TPCANStatus result;
StringBuilder^ strMsg;
String^ strBuffer;

strMsg = gcnew StringBuilder(256);

// The path for the Log file is set.
// Note that this parameter is set using the
// default Channel (PCAN_NONEBUS)
//
strBuffer = "C:\\Users\\Admin\\Desktop";
result = PCANBasic::SetValue(PCANBasic::PCAN_NONEBUS)
if (result != TPCANStatus::PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the
    //
    PCANBasic::GetErrorText(result, 0, strMsg);
    MessageBox::Show(strMsg->ToString());
}
else
    MessageBox::Show("Log path was successfully set");
```

Visual Basic:

```vbnet
Dim result As TPCANStatus
Dim strMsg As StringBuilder
Dim strBuffer As String
```
strMsg = New StringBuilder(256)

' The path for the Log file is set.
' Note that this parameter is set using the
' default Channel (PCAN_NONEBUS)
,'

strBuffer = "C:\Users\Admin\Desktop"
result = PCANBasic.SetValue(PCANBasic.PCAN.NONEBUS)
If result <> TPCANStatus.PCAN_ERROR_OK Then
    ' An error occurred, get a text describing the
    '
    PCANBasic.GetErrorText(result, 0, strMsg)
    MessageBox.Show(strMsg.ToString())
Else
    MessageBox.Show("Log path was successfully set")
End If

Pascal OO:

var
    result : TPCANStatus;
    strMsg: array [0..256] of Char;
    strBuffer: string;
begin
    // The path for the Log file is set.
    // Note that this parameter is set using the
    // default Channel (PCAN_NONEBUS)
    //
    strBuffer := 'C:\Users\Keneth\Desktop';
    result := TPCANBasic.SetValue(TPCANBasic.PCAN_NONEBUS)
    If (result <> PCAN_ERROR_OK) Then
        begin
            // An error occurred, get a text describing
            //
            TPCANBasic.GetErrorText(result, 0, strMsg)
            MessageBox(0, strMsg, 'Error',MB_OK);
        end
    end
else
MessageBox(0, 'Log path was successfully set')

See Also

GetValue
TPCANParameter
Parameter Value Definitions

Plain function Version: CAN_SetValue
SetValue(TPCANHandle, TPCANParameter, UInt32, UInt32)

Sets a configuration or information numeric value within a PCAN Channel.

Syntax

**Pascal OO**

```pascal
class function SetValue(
  Channel: TPCANHandle;
  Parameter: TPCANParameter;
  NumericBuffer: PLongWord;
  BufferLength: LongWord
): TPCANStatus; overload;
```

**C#**

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_SetValue")]
public static extern TPCANStatus SetValue(
  [MarshalAs(UnmanagedType.U1)] TPCANHandle Channel,
  [MarshalAs(UnmanagedType.U1)] TPCANParameter Parameter,
  ref UInt32 NumericBuffer,
  UInt32 BufferLength);
```

**C++ / CLR**

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_SetValue")]
static TPCANStatus SetValue(
  [MarshalAs(UnmanagedType::U1)] TPCANHandle Channel,
  [MarshalAs(UnmanagedType::U1)] TPCANParameter Parameter,
  UInt32 BufferLength);
```
```csharp
UInt32 %NumericBuffer,
UInt32 BufferLength);
```

### Visual Basic

```vbnet
<DllImport("PCANBasic.dll", EntryPoint:="CAN_SetValue")>
Public Shared Function SetValue(_
    <MarshalAs(UnmanagedType.U1)> _
    ByVal Channel As TPCANHandle, _
    <MarshalAs(UnmanagedType.U1)> _
    ByVal Parameter As TPCANParameter, _
    ByRef NumericBuffer As UInt32, _
    ByVal BufferLength As UInt32) As TPCANStatus
End Function
```

### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the value to be set (see TPCANParameter).</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 TPCANStatus code. PCAN_ERROR_OK is returned on success. The typical errors in case of failure are:

- **PCAN_ERROR_ILLPARAMVAL**: Indicates that the
Parameters passed to the method are invalid. Check the parameter 'NumericBuffer'; it should point to an integer buffer.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PCAN_ERROR_CAUTION:</strong></td>
<td>The configuration of a parameter failed due to a no more existing channel. The parameter has been reset on all existing channels.</td>
</tr>
<tr>
<td><strong>PCAN_ERROR_INITIALIZE:</strong></td>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td><strong>PCAN_ERROR_IILLPARAMTYPE:</strong></td>
<td>Indicates that the requested information is not available for the given PCAN Channel. Check the value of 'Parameter'; some values are not available for all PCAN-Channels or cannot be set.</td>
</tr>
<tr>
<td><strong>PCAN_ERROR_Illoperation:</strong></td>
<td>An underlying process that is generated by a call to this method with the current parameters, is temporarily not allowed. The configuration in relation to the used TPCANParameter must be checked.</td>
</tr>
</tbody>
</table>

**Remarks**

Use the method **SetValue** to set configuration information or
environment values of a PCAN Channel as parameters like the Message Filter and values like a custom entry in the log file of PCAN-Basic. Take in account that not all parameters are supported for all PCAN-Channels. The access's type of the parameters can also be different.

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 PCAN_USBBUS1 to close the message filter. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

C#:

```csharp
TPCANStatus result;
StringBuilder strMsg;
UInt32 iBuffer;

strMsg = new StringBuilder(256);

// The message filter is closed
//
// iBuffer = PCANBasic.PCAN_FILTER_CLOSE;
result = PCANBasic.SetValue(PCANBasic.PCAN_USBBUS1, iBuffer);
if (result != TPCANStatus.PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    //
    PCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox.Show(strMsg.ToString());
}
else
```
MessageBox.Show("The filter was successfully closed\n\nC++/CLR:

```c++
TPCANStatus result;
StringBuilder^ strMsg;
UInt32 iBuffer;

strMsg = gcnew StringBuilder(256);

// The message filter is closed
//
iBuffer = PCANBasic::PCAN_FILTER_CLOSE;
result = PCANBasic::SetValue(PCANBasic::PCAN_USBBUS1, iBuffer);
if (result != TPCANStatus::PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    //
    PCANBasic::GetErrorText(result, 0, strMsg);
    MessageBox::Show(strMsg->ToString());
}
else
    MessageBox::Show("The filter was successfully closed\n```

Visual Basic:

```vbnet
Dim result As TPCANStatus
Dim strMsg As StringBuilder
Dim iBuffer As UInt32

strMsg = New StringBuilder(256)

' The message filter is closed
'
iBuffer = PCANBasic.PCAN_FILTER_CLOSE
result = PCANBasic.SetValue(PCANBasic.PCAN_USBBUS1
If result <> TPCANStatus.PCAN_ERROR_OK Then
    ' An error occurred, get a text describing the error
```
PCANBasic.GetErrorText(result, 0, strMsg)
MessageBox.Show(strMsg.ToString)

Else

MessageBox.Show("The filter was successfully closed")

End If

Pascal OO:

var
result : TPCANStatus;
strMsg: array [0..256] of Char;
iBuffer: LongWord;

begin

// The message filter is closed

//
iBuffer := TPCANBasic.PCAN_FILTER_CLOSE;
result := TPCANBasic.SetValue(TPCANBasic.PCAN_If (result <> PCAN_ERROR_OK) Then

begin

// An error occurred, get a text describing it

//
TPCANBasic.GetErrorText(result, 0, strMsg)
MessageBox(0, strMsg, 'Error', MB_OK);

end

else

MessageBox(0, 'The filter was successfully closed')

See Also

GetValue
TPCANParameter
Parameter Value Definitions

Plain function Version: CAN_SetValue
Set Value (TPCANHandle, TPCANParameter, UInt64, UInt32)

Sets a configuration or information numeric value within a PCAN Channel.

Syntax

**Pascal OO**

```pascal
class function SetValue(
    Channel: TPCANHandle;
    Parameter: TPCANParameter;
    NumericBuffer: PUInt64;
    BufferLength: LongWord
) : TPCANStatus; overload;
```

**C#**

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CANSetValue")
public static extern TPCANStatus SetValue(
    [MarshalAs(UnmanagedType.U1)] TPCANHandle Channel,
    [MarshalAs(UnmanagedType.U1)] TPCANParameter Parameter,
    ref UInt64 NumericBuffer,
    UInt32 BufferLength);
```

**C++ / CLR**

```cpp
[DllImport("PCANBasic.dll", EntryPoint = "CANSetValue")
static TPCANStatus SetValue(
    [MarshalAs(UnmanagedType::U1)] TPCANHandle Channel,
    [MarshalAs(UnmanagedType::U1)] TPCANParameter Parameter,
    TPCANParameter Parameter,
    UInt32 BufferLength);
```

Visual Basic

<DllImport("PCANBasic.dll", EntryPoint:="CAN_SetValue")>
Public Shared Function SetValue(
    ByVal Channel As TPCANHandle,
    ByVal Parameter As TPCANParameter,
    ByRef NumericBuffer As UInt64,
    ByVal BufferLength As UInt32) As TPCANStatus
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the value to be set (see TPCANParameter).</td>
</tr>
<tr>
<td>NumericBuffer</td>
<td>The buffer containing the 64-bit 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 TPCANStatus code. PCAN_ERROR_OK is returned on success. The typical errors in case of failure are:

<p>| PCAN_ERROR_ILLPARAMVAL: | Indicates that the |</p>
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_CAUTION:</td>
<td>The configuration of a parameter failed due to a no more existing channel. The parameter has been reset on all existing channels.</td>
</tr>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLPARAMTYPE:</td>
<td>Indicates that the requested information is not available for the given PCAN Channel. Check the value of 'Parameter'; some values are not available for all PCAN-Channels or cannot be set.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLOPERATION:</td>
<td>An underlying process that is generated by a call to this method with the current parameters, is temporarily not allowed. The configuration in relation to the used <strong>TPCANParameter</strong> must be checked.</td>
</tr>
</tbody>
</table>

Remarks
Use the method **SetVal**e to set configuration information or environment values of a PCAN Channel as parameters like an Acceptance Filter. Take in account that not all parameters are supported for all PCAN-Channels. The access's type of the parameters can also be different.

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 **SetVal**e on the channel PCAN_USBBUS1 to set a message filter as a 11-bit acceptance code and mask, allowing only IDs from 0x100 to 0x103. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

**C#**:

```csharp
TPCANStatus result;
StringBuilder strMsg;
UInt64 i64Buffer;

strMsg = new StringBuilder(256);

// The acceptance code and mask are packed as 64-b
//
i64Buffer = 0x100;  // Acceptance code
i64Buffer <<= 32;
i64Buffer |= 0x003;  // Acceptance mask
result = PCANBasic.SetValue(PCANBasic.PCAN_USBBUS1
if (result != TPCANStatus.PCAN_ERROR_OK)
{
    // An error occurred, get a text describing th
    //
    PCANBasic.GetErrortext(result, 0, strMsg);
```
C++/CLR:

```c++
TPCANStatus result;
StringBuilder^ strMsg;
UInt64 i64Buffer;

strMsg = gcnew StringBuilder(256);

// The acceptance code and mask are packed as 64-bit
//
i64Buffer = 0x100; // Acceptance code
i64Buffer <<= 32;
i64Buffer |= 0x003; // Acceptance mask
result = PCANBasic::SetValue(PCANBasic::PCAN_USB_BU,
   if (result != TPCANStatus::PCAN_ERROR_OK)
   {
      // An error occurred, get a text describing the
      //
      PCANBasic::GetErrorText(result, 0, strMsg);
      MessageBox::Show(strMsg->ToString());
   }
else
   MessageBox::Show("The 11-bit acceptance filter was successfully configured.");
```

Visual Basic:

```vbnet
Dim result As TPCANStatus
Dim strMsg As StringBuilder
Dim i64Buffer As UInt64

strMsg = New StringBuilder(256)

' The acceptance code and mask are packed as 64-bit
```
i64Buffer = &H100  ' Acceptance code
i64Buffer <<= 32
i64Buffer = i64Buffer Or &H3  ' Acceptance mask

result = PCANBasic.SetValue(PCANBasic.PCAN_USBBUS1
If result <> TPCANStatus.PCAN_ERROR_OK Then
  ' An error occurred, get a text describing the
  PCANBasic.GetErrorText(result, 0, strMsg)
  MessageBox.Show(strMsg.ToString)
Else
  MessageBox.Show("The 11-bit acceptance filter
End If

Pascal OO:

var
  result : TPCANStatus;
  strMsg: array [0..256] of Char;
  i64Buffer: UInt64;
begin
  // The acceptance code and mask are packed as
  //
  i64Buffer := $100;  // Acceptance c
  i64Buffer := i64Buffer shl 32;
  i64Buffer := i64Buffer Or $3;  // Acceptance m
result := PCANBasic.SetValue(TPCANBasic.PCAN_ If (result <> PCAN_ERROR_OK) Then
begin
  // An error occurred, get a text describing
  //
  TPCANBasic.GetErrorText(result, 0, strMsg)
  MessageBox(0, strMsg, 'Error',MB_OK);
end
else
  MessageBox(0, 'The 11-bit acceptance filte
end;
See Also

getValue
TPCANParameter
Parameter Value Definitions

Plain function Version: CAN_SetValue
Set\text{Value}(\text{TPCANHandle}, \text{TPCANParameter}, \text{Object})

Sets a configuration or information value within a PCAN Channel.

\textbf{Syntax}

\begin{verbatim}
def \text{SetValue}(self, Channel, Parameter, Buffer)
\end{verbatim}

\textbf{Parameters}

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see \text{TPCANHandle}).</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the value to be set (see \text{TPCANParameter}).</td>
</tr>
<tr>
<td>Buffer</td>
<td>The buffer containing the value to be set.</td>
</tr>
</tbody>
</table>

\textbf{Returns}

The return value is a \text{TPCANStatus} code. PCAN\_ERROR\_OK is returned on success. The typical errors in case of failure are:

| PCAN\_ERROR\_ILLPARAMVAL: | Indicates that the parameters passed to the method are invalid. Check |
the parameter 'Buffer'; it should point to a buffer of a type which is accepted by the parameter being configured.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_CAUTION:</td>
<td>The configuration of a parameter failed due to a no more existing channel. The parameter has been reset on all existing channels.</td>
</tr>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td>PCAN_ERROR_I LLPARAMTYPE:</td>
<td>Indicates that the requested information is not available for the given PCAN Channel. Check the value of 'Parameter'; some values are not available for all PCAN-Channels or cannot be set.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLOPERATION:</td>
<td>An underlying process that is generated by a call to this method with the current parameters, is temporarily not allowed. The configuration in relation to the used TPCANParameter must be checked.</td>
</tr>
</tbody>
</table>

Remarks

Use the method SetValue to set configuration information or
environment values of a PCAN Channel as parameters like the Message Filter and values like a custom entry in the log file of PCAN-Basic. Take in account that not all parameters are supported for all PCAN-Channels. The access's type of the parameters can also be different.

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

Python Notes

- Class-Method: Unlike the .NET Framework, under Python a variable has to be instantiated with an object of type PCANBasic in order to use the API functionality.
- Python's first argument convention: Under Python, 'self' is a parameter that is automatically included within the call of this method, within a PCANBasic object and hasn't to be indicated in a method call. This parameter represents the calling object itself.

Example

The following example shows the use of the method SetValue on the channel PCAN_NONEBUS to set (and activate) the path for the log file of a PCAN-Basic's debug session. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: It is not needed to have an initialized PCAN channel for using the Log functionality.

Python:

```python
# The path for the Log file is set.
# Note that this parameter is set using the
# default Channel (PCAN_NONEBUS)
#
strBuffer = "C:\Users\Admin\Desktop"
result = objPCAN.SetValue(PCAN_NONEBUS, PCAN_LOG_LOCATION)
if result != PCAN_ERROR_OK:
```
# An error occurred, get a text describing the error

```python
result = objPCAN.GetErrorText(result)
print result[1]
else:
    print "Log path was successfully set"
```

See Also

- GetValue
- TPCANParameter
- Parameter Value Definitions

Plain function Version: CAN_SetValue
FilterMessages

Configures the reception filter.

Syntax

Pascal OO

```pascal
class function FilterMessages(
    Channel: TPCANHandle;
    FromID: LongWord;
    ToID: LongWord;
    Mode: TPCANMode
): TPCANStatus;
```

C#

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_FilterMessages")
public static extern TPCANStatus FilterMessages([MarshalAs(UnmanagedType.U1)]
    TPCANHandle Channel,
    UInt32 FromID,
    UInt32 ToID,
    [MarshalAs(UnmanagedType.UnmanagedType:U1)]
    TPCANMode Mode);
```

C++/CLR

```cpp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_FilterMessages")
static TPCANStatus FilterMessages(
    [MarshalAs(UnmanagedType.UnmanagedType::U1)]
    TPCANHandle Channel,
    UInt32 FromID,
    UInt32 ToID,
    [MarshalAs(UnmanagedType.UnmanagedType::U1)]
    TPCANMode Mode);
```
Visual Basic

<DllImport("PCANBasic.dll", EntryPoint:="CAN_FilterMessages")>
Public Shared Function FilterMessages(_
    <MarshalAs(UnmanagedType.U1)> ByVal Channel As TPCANHandle, _
    ByVal FromID As UInt32, _
    ByVal ToID As UInt32, _
    <MarshalAs(UnmanagedType.U1)> ByVal Mode As TPCANMode) As TPCANStatus
End Function

Python

def FilterMessages(
    self,
    Channel,
    FromID,
    ToID,
    Mode)

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>FromID</td>
<td>The lowest CAN ID wanted to be received.</td>
</tr>
<tr>
<td>ToID</td>
<td>The highest CAN ID wanted to be received.</td>
</tr>
<tr>
<td>Mode</td>
<td>The type of the filter being set (see TPCANMode).</td>
</tr>
</tbody>
</table>
Returns

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

| PCAN_ERROR_INITIALIZE: | Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application. |

Remarks

Note that after a PCAN Channel is initialized, the status of its filter is fully opened. According with the current filter status, calling this method causes the following behavior:

- Filter status is PCAN_FILTER_OPEN: The filter is automatically closed and then configured with the given range of IDs passed to this function [FromID, ToID].
- Filter status is PCAN_FILTER_CLOSE: The filter is set to the given range of IDs passed to this function [FromID, ToID].
- Filter status is PCAN_FILTER_CUSTOM: The filter is expanded with the given range of IDs [FromID, ToID]. If a smaller or different range is required than a range that has been configured before, the filter has to be closed first before calling the method FilterMessages. To do this use the method SetValue.

The parameter 'Mode' indicates which kind of ID is being used to register the new filter range. There are two possible values, Standard (11-bit identifier) or Extended (29-bit identifier). Standard frames are using the bit positions 28 to 18 of the Acceptance Mask/Code registers in the SJA1000 CAN controller. Drivers for 82C200 CAN controllers have to shift the bits down to positions 10 to 0.

Take in account that configuring the message filter cause the CAN controller to enter the Reset state. This will affect other applications that communicate with the same PCAN hardware.

Notes:
1. There is only one filter for standard and extended CAN messages. It seems that the ID from a standard message uses the most significant 11 bits (bit 18 to 28) of the 29 bits. I.e. the standard ID 400h is also received by indicating an extended ID 10000000h. For this reason it is not recommended to mix standard and extended filters, since it can increase the risk of receiving unwanted messages.

2. Multiple calls of **FilterMessages** expand the reception filter.

3. It is not guaranteed that an application only receives CAN messages in the range of **FromID** to **ToID**. This is caused by the operating principle of the SJA1000's acceptance filter. See also Philips Data Sheet "SJA1000 Stand-alone CAN-controller".

**Python Notes**

- Class-Method: Unlike the .NET Framework, under Python a variable has to be instantiated with an object of type [PCANBasic](#) in order to use the API functionality.
- Python's first argument convention: Under Python, 'self' is a parameter that is automatically included within the call of this method, within a [PCANBasic](#) object and hasn't to be indicated in a method call. This parameter represents the calling object itself.

**Example**

The following example shows the use of the method FilterMessages on the channel PCAN_USBBUS1 to receive a custom range of IDs. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

**C#**:

```csharp
TPCANStatus result;
StringBuilder strMsg;
UInt32 iBuffer;

strMsg = new StringBuilder(256);
```
// The message filter is closed first to ensure the new range of IDs.

iBuffer = PCANBasic.PCAN_FILTER_CLOSE;
result = PCANBasic.SetValue(PCANBasic.PCAN_USBBUS1, iBuffer);

if (result != TPCANStatus.PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    PCANBasic.GetErrorText(result, 0, strMsg);
    MessageBox.Show(strMsg.ToString());
}
else
{
    // The message filter is configured to receive IDs 2, 3, 4, and 5 on the PCAN-USB, Channel 1
    result = PCANBasic.FilterMessages(PCANBasic.PCAN_USBBUS1);
    if (result != TPCANStatus.PCAN_ERROR_OK)
    {
        // An error occurred, get a text describing the error
        PCANBasic.GetErrorText(result, 0, strMsg);
        MessageBox.Show(strMsg.ToString());
    }
    else
    {
        MessageBox.Show("Filter successfully configured for IDs 2, 3, 4 and 5");
    }
}

C++/CLR:

TPCANStatus result;
StringBuilder^ strMsg;
UInt32 iBuffer;

strMsg = gcnew StringBuilder(256);

// The message filter is closed first to ensure the new range of IDs.
Visual Basic:

```vbnet
dim result as TPCANStatus
Dim strMsg as StringBuilder
Dim iBuffer as UInt32

strMsg = New StringBuilder(256)

' The message filter is closed first to ensure that
'iBuffer = PCANBasic.PCAN_FILTER_CLOSE
result = PCANBasic.SetValue(PCANBasic.PCAN_USBBUS1
```
If result <> TPCANStatus.PCAN_ERROR_OK Then
  ' An error occurred, get a text describing the error.'
  PCANBasic.GetErrorText(result, 0, strMsg)
  MessageBox.Show(strMsg.ToString)
Else
  ' The message filter is configured to receive IDs 2, 3, 4 and 5 on the PCAN-USB, Channel 1
  result = PCANBasic.FilterMessages(PCANBasic.PCAN_USBBUS1)
If result <> TPCANStatus.PCAN_ERROR_OK Then
  ' An error occurred, get a text describing the error.'
  PCANBasic.GetErrorText(result, 0, strMsg)
  MessageBox.Show(strMsg.ToString)
Else
  MessageBox.Show("Filter successfully configured for IDs 2, 3, 4 and 5")
End If
End If

Pascal OO:

var
  result : TPCANStatus;
  strMsg: array [0..256] of Char;
  iBuffer: LongWord;
begin
  // The message filter is closed first to ensure the reception of the new range of IDs.
  // iBuffer := PCANBasic.PCAN_FILTER_CLOSE;
  result := PCANBasic.SetValue(PCANBasic.PCAN_USBBUS1)
If (result <> PCAN_ERROR_OK) Then
  begin
    // An error occurred, get a text describing the error.
    //
    TPCANBasic.GetErrorText(result, 0, strMsg)
    MessageBox(0, strMsg, 'Error', MB_OK);
  end
else

begin
// The message filter is configured to receive the IDs 2, 3, 4, and 5 on the PCAN-USB, Channel 1
result := TPCANBasic.FilterMessages(TPCANBasic.PCAN_USBBUS1)
If (result <> PCAN_ERROR_OK) Then begin
  // An error occurred, get a text describing the error and show it
  TPCANBasic.GetErrorText(result, 0, strMsg);
  MessageBox(0, strMsg, 'Error', MB_OK);
end else
  MessageBox(0, 'Filter successfully configured for IDs 2, 3, 4, and 5');
end;

Python:

# The message filter is closed first to ensure the reception of the new range of IDs.
# result = objPCAN.SetValue(PCAN_USBBUS1, PCAN_MESSAGE_FILTER)
if result != PCAN_ERROR_OK:
  # An error occurred, get a text describing the error
  result = objPCAN.GetErrorText(result)
  print result[1]
else:
  # The message filter is configured to receive the IDs 2, 3, 4, and 5
  result = objPCAN.FilterMessages(PCAN_USBBUS1, 2, 5)
  if result != PCAN_ERROR_OK:
    # An error occurred, get a text describing the error
    result = objPCAN.GetErrorText(result)
    print result[1]
  else:
    print "Filter successfully configured for IDs 2, 3, 4, and 5"
See Also

SetValue

Plain function Version: CAN_FilterMessages
GetErrorText

Gets a descriptive text for an error code.

**Overloads**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetErrorText(TPCANStatus, UInt16, String)</td>
<td>Gets a descriptive text for an error code.</td>
</tr>
<tr>
<td>GetErrorText(TPCANStatus, int)</td>
<td>Gets a descriptive text for an error code.</td>
</tr>
</tbody>
</table>
GetErrorText(TPCANStatus, UInt16, String)

Gets a descriptive text for an error code.

Syntax

**Pascal OO**

```pascal
class function GetErrorText(
    AnError: TPCANStatus;
    Language: Word;
    StringBuffer: PChar
): TPCANStatus;
```

**C#**

```csharp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_GetErrorText")
public static extern TPCANStatus GetErrorText(
    [MarshalAs(UnmanagedType.U4)] TPCANStatus AnError,
    UInt16 Language,
    StringBuilder StringBuffer);
```

**C++ / CLR**

```cpp
[DllImport("PCANBasic.dll", EntryPoint = "CAN_GetErrorText")
static TPCANStatus GetErrorText(
    [MarshalAs(UnmanagedType.U4)] TPCANStatus AnError,
    UInt16 Language,
    StringBuilder^ StringBuffer);
```

**Visual Basic**

```vb
<DllImport("PCANBasic.dll", EntryPoint:="CAN_GetErrorText")
Public Shared Function GetErrorText( _
    <MarshalAs(UnmanagedType.U4)> _
```

```
ByVal AnError As TPCANStatus, _
ByVal Language As UInt16, _
ByVal StringBuffer As StringBuilder) As TPCANStatus
End Function

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnError</td>
<td>A TPCANStatus error code.</td>
</tr>
<tr>
<td>Language</td>
<td>Indicates a &quot;Primary language ID&quot;.</td>
</tr>
<tr>
<td>StringBuffer</td>
<td>A buffer for a null-terminated char array.</td>
</tr>
</tbody>
</table>

Returns

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

| PCAN_ERROR_IILLPARAMVAL: | Indicates that the parameters passed to the method are invalid. Check the parameter 'Buffer'; it should point to a char array, big enough to allocate the text for the given error code. |

Remarks

The "Primary language IDs" are codes used by Windows OS from Microsoft, to identify a human language. The PCAN-Basic API currently support the following languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Primary Language ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Code</td>
</tr>
<tr>
<td>------------------</td>
<td>------</td>
</tr>
<tr>
<td>Neutral (System dependant)</td>
<td>00h (0)</td>
</tr>
<tr>
<td>English</td>
<td>09h (9)</td>
</tr>
<tr>
<td>German</td>
<td>07h (7)</td>
</tr>
<tr>
<td>French</td>
<td>0Ch (12)</td>
</tr>
<tr>
<td>Italian</td>
<td>10h (16)</td>
</tr>
<tr>
<td>Spanish</td>
<td>0Ah (10)</td>
</tr>
</tbody>
</table>

**Note:** If the buffer is too small for the resulting text, the error PCAN_ERROR_ILLPARAMVAL is returned. Even when only short texts are being currently returned, a text within this function can have a maximum of 255 characters. For this reason it is recommended to use a buffer with a length of at least 256 bytes.

**Example**

The following example shows the use of the method `GetErrorText` to get the description of an error. The language of the description's text will be the same used by the operating system (if its language is supported; otherwise English is used).

**C#**

```csharp
TPCANStatus result;
StringBuilder strMsg;

strMsg = new StringBuilder(256);

// Gets the description text for PCAN_ERROR_INITIATE
//
result = PCANBasic.GetErrorText(TPCANStatus.PCAN_ERROR_INITIATE)
if (result != TPCANStatus.PCAN_ERROR_OK)
    // An error occurred, show a message indicating
    //
    MessageBox.Show("Error when recovering Error-Code");
else
```
MessageBox.Show(strMsg.ToString());

**C++/CLR:**

```c++
TPCANStatus result;
StringBuilder^ strMsg;

strMsg = gcnew StringBuilder(256);

// Gets the description text for PCAN_ERROR_INITIALIZE
// result = PCANBasic::GetErrorText(TPCANStatus::PCAN_ERROR_INITIALIZE)
if (result != TPCANStatus::PCAN_ERROR_OK)
    // An error occurred, show a message indicating it
    MessageBox::Show("Error when recovering Error-Code's description");
else
    MessageBox::Show(strMsg->ToString());
```

**Visual Basic:**

```vbnet
Dim result As TPCANStatus
Dim strMsg As StringBuilder

strMsg = New StringBuilder(256)

' Gets the description text for PCAN_ERROR_INITIALIZE
result = PCANBasic.GetErrorText(TPCANStatus.PCAN_ERROR_INITIALIZE)
If result <> TPCANStatus.PCAN_ERROR_OK Then
    ' An error occurred, show a message indicating it
    MessageBox.Show("Error when recovering Error-Code's description");
Else
    MessageBox.Show(strMsg.ToString)
End If
```

**Pascal OO:**

```pascal
Dim result As TPCANStatus
Dim strMsg As StringBuilder

strMsg := New StringBuilder(256)

' Gets the description text for PCAN_ERROR_INITIALIZE
result := PCANBasic.GetErrorText(TPCANStatus.PCAN_ERROR_INITIALIZE)
If result <> TPCANStatus.PCAN_ERROR_OK Then
    ' An error occurred, show a message indicating it
    MessageBox.Show("Error when recovering Error-Code's description");
Else
    MessageBox.Show(strMsg.ToString)
End If
```
var
result : TPCANStatus;
strMsg: array [0..256] of Char;
begin

// Gets the description text for PCAN_ERROR_INITIALIZE
//
result := TPCANBasic.GetErrorText(PCAN_ERROR_INITIALIZE);
If (result <> PCAN_ERROR_OK) Then
  // An error occurred, show a message indicating it
  MessageBox(0, Error when recovering Error-
else
  MessageBox(0, strMsg,'Success', MB_OK);

See Also

Primary Language ID

Plain function Version: CAN_GetErrorText
GetErrorText(TPCANStatus, int)

Gets a descriptive text for an error code.

Syntax

```python
def GetErrorText(self, Error, Language = 0)
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>A TPCANStatus error code.</td>
</tr>
<tr>
<td>Language</td>
<td>Indicates a &quot;Primary language ID&quot;.</td>
</tr>
</tbody>
</table>

Returns

The return value is a 2-touple. The order of the returned values is as follow:

[0]: The method's return value as a TPCANStatus code. PCAN_ERROR_OK is returned on success. The typical errors in case of failure are:

PCAN_ERROR_ILLPARAMVAL: Indicates that the parameters passed to the method are invalid. Check the parameter 'Buffer'; it should point to a char array, big enough to allocate the text for the given
[1]: The text corresponding to the given TPCANStatus code.

### Remarks

The "Primary language IDs" are codes used by Windows OS from Microsoft, to identify a human language. The PCAN-Basic API currently support the following languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Primary Language ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neutral (System dependant)</td>
<td>00h (0)</td>
</tr>
<tr>
<td>English</td>
<td>09h (9)</td>
</tr>
<tr>
<td>German</td>
<td>07h (7)</td>
</tr>
<tr>
<td>French</td>
<td>0Ch (12)</td>
</tr>
<tr>
<td>Italian</td>
<td>10h (16)</td>
</tr>
<tr>
<td>Spanish</td>
<td>0Ah (10)</td>
</tr>
</tbody>
</table>

### Python Notes

- Class-Method: Unlike the .NET Framework, under Python a variable has to be instantiated with an object of type PCANBasic in order to use the API functionality.
- Python's first argument convention: Under Python, 'self' is a parameter that is automatically included within the call of this method, within a PCANBasic object and hasn't to be indicated in a method call. This parameter represents the calling object itself.

### Example

The following example shows the use of the method GetErrorText to get the description of an error. The language of the description's text will be in Spanish.

**Python:**

...
# Gets the description text for PCAN_ERROR_INITIALIZE

```python
objPCAN = PCANBasic()
result = objPCAN.GetErrorText(PCAN_ERROR_INITIALIZE)
if result[0] != PCAN_ERROR_OK:
    # An error occurred, show a message indicating
    #
    print "Error when recovering Error-Code's desc"
else:
    print result[1]
```

See Also

- [Primary Language ID](#)

Plain function Version: [CAN_GetErrorText](#)
The functions of the PCAN-Basic API are divided in 4 groups of functionality:

## Connection

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_Initialize</td>
<td>Initializes a PCAN Channel.</td>
</tr>
<tr>
<td>CAN_InitializeFD</td>
<td>Initializes a <strong>FD capable</strong> PCAN Channel.</td>
</tr>
<tr>
<td>CAN_Uninitialize</td>
<td>Uninitializes a PCAN Channel.</td>
</tr>
</tbody>
</table>

## Configuration

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_SetValue</td>
<td>Sets a configuration or information value within a PCAN Channel.</td>
</tr>
<tr>
<td>CAN_FilterMessages</td>
<td>Configures the message's reception filter of a PCAN Channel.</td>
</tr>
</tbody>
</table>

## Information

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_GetValue</td>
<td>Retrieves information from a PCAN Channel.</td>
</tr>
<tr>
<td>CAN_GetStatus</td>
<td>Retrieves the current BUS status of a PCAN Channel.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CAN_GetErrorText</td>
<td>Returns a descriptive text for an error code.</td>
</tr>
</tbody>
</table>

**Communication**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_Read</td>
<td>Reads a CAN message from the receive queue of a PCAN Channel.</td>
</tr>
<tr>
<td>CAN_ReadFD</td>
<td>Reads a CAN message from the receive queue of a <strong>FD capable</strong> PCAN Channel.</td>
</tr>
<tr>
<td>CAN_Write</td>
<td>Transmits a CAN message using a connected PCAN Channel.</td>
</tr>
<tr>
<td>CAN_WriteFD</td>
<td>Transmits a CAN message using a connected <strong>FD capable</strong> PCAN Channel.</td>
</tr>
<tr>
<td>CAN_Reset</td>
<td>Resets the receive and transmit queues of a PCAN Channel.</td>
</tr>
</tbody>
</table>
CAN_Initialize

Initializes a PCAN Channel.

Syntax

```cpp
#ifdef __cplusplus
#define _DEF_ARG = 0  // Using of default arguments
#else
#define _DEF_ARG
#endif

TPCANStatus __stdcall CAN_Initialize(
    TPCANHandle Channel,
    TPCANBaudrate Btr0Btr1,
    TPCANType HwType _DEF_ARG,
    DWORD IOPort _DEF_ARG,
    WORD Interrupt _DEF_ARG
);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>Btr0Btr1</td>
<td>The speed for the communication (BTR0BTR1 code).</td>
</tr>
<tr>
<td>HwType</td>
<td>The type of the Non-Plug-and-Play hardware and its operation</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>IOPort</strong></td>
<td>The I/O address for the parallel port of the Non-Plug-and-Play hardware.</td>
</tr>
<tr>
<td><strong>Interrupt</strong></td>
<td>The Interrupt number of the parallel port of the Non-Plug-and-Play hardware.</td>
</tr>
</tbody>
</table>

**Returns**

The return value is a [TPCANStatus](#) code. PCAN_ERROR_OK is returned on success. The typical errors in case of failure are:

<table>
<thead>
<tr>
<th>PCAN_ERROR_CAUTION:</th>
<th>Indicates that the channel has been initialized but at a different bit rate as the given one.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_I LLHANDLE:</td>
<td>Indicates that the desired PCAN Channel is not valid. Check the list of <a href="#">valid Channels</a>.</td>
</tr>
<tr>
<td>PCAN_ERROR_I LLHW:</td>
<td>Indicates that the desired PCAN Channel is not available.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLOPERATION:</td>
<td>Indicates that an action cannot be executed due to the state of the hardware. Possible causes are:</td>
</tr>
<tr>
<td></td>
<td>• The desired PCAN-Channel is a LAN Channel, which uses a different bit rate than the specified.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PCAN_ERROR_REGTEST:</td>
<td>(Not-Plug-And-Play Only) Indicates a problem with hardware registration, normally due to wrong values in the parameters 'HwType', 'IOPort' and 'Interrupt'.</td>
</tr>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the desired PCAN channel cannot be connected because it is already in use (PCAN-Basic / PCAN-Light environment).</td>
</tr>
<tr>
<td>PCAN_ERROR_NETINUSE:</td>
<td>Indicates that the desired PCAN-Channel is being used with a different bit rate (PCAN-View).</td>
</tr>
<tr>
<td>PCAN_ERROR_HWINUSE:</td>
<td>Indicates that the desired PCAN-Channel is being used (CanApi2 connection).</td>
</tr>
<tr>
<td>PCAN_ERROR_NODRIVER:</td>
<td>The driver needed for connecting the desired PCAN Channel is not loaded.</td>
</tr>
</tbody>
</table>

Remarks

**Note on correspondence of Functions:**

A Channel that is initialized using CAN_Initialize must use CAN_Read and CAN_Write for communication. Calling CAN_ReadFD and/or CAN_WriteFD will result in a **PCAN_ERROR_ILLOPERATION** error.

As indicated by its name, the CAN_Initialize function initiates a PCAN Channel, preparing it for communicate within the CAN bus connected
to it. Calls to the API functions will fail if they are used with a Channel handle, different than PCAN_NONEBUS, that has not been initialized yet. Each initialized channel should be released when it is not needed anymore.

Initializing a PCAN Channel means:

- to reserve the Channel for the calling application/process.
- to allocate channel resources, like receive and transmit queues.
- to register/connect the Hardware denoted by the channel handle.
- to check and adapt the bus speed, if the Channel is already in use. (Only if the Channel was pre-configured as Bitrate Adapting; see: Bitrate-Adapting Parameter).
- to set the channel in Listen-Only mode. (Only if the channel was pre-configured as Listen-Only; see: Listen-Only Parameter).
- to open the messages filter for the application/process.
- to set-up the default values of the different parameters (See CAN_GetValue).
- to set the Receive Status of the channel. (Pre-configured value; see: Receive Status Parameter).

Different than the PCAN-Light API, the Initialization process will fail if an application try to initialize a PCAN-Channel that has been initialized already within the same process.

The PCAN-Basic API use the same function for initializations of both, Plug-And-Play and Not-Plug-And-Play hardware. The CAN_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.

Take in consideration that initializing a channel causes a reset of the CAN hardware, when the bus status is other than OK. In this way errors like BUSOFF, BUSHEAVY, and BUSLIGHT, are removed.

**PCAN-LAN Channels**

A PCAN-LAN channel doesn't allow changing the bit rate using PCAN-Basic. In order to connect a PCAN-LAN Channel it is necessary to know the bit rate of the PCAN-Gateway device that is
represented by that channel. If the bit rate is not known, the parameter **Bitrate-Adapting** should be used.

### Example

The following example shows the initialize and uninitialize processes for a Plug-And-Play channel (channel 2 of a PCAN-PCI hardware) and for a Not-Plug-And-Play channel (channel 1 of the PCAN-DNG). In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

**C++:**

```cpp
TPCANStatus result;
char strMsg[256];

// The Plug & Play Channel (PCAN-PCI) is initialize
result = CAN_Initialize(PCAN_PCIBUS2, PCAN_BAUD_500);
if(result != PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the
    //
    CAN_GetErrorText(result, 0, strMsg);
    MessageBox(strMsg);
} else
    MessageBox("PCAN-PCI (Ch-2) was initialized");

// The Not Plug & Play Channel (PCAN-Dongle) is in
result = CAN_Initialize(PCAN_DNGBUS1, PCAN_BAUD_500);
if(result != PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the
    //
    CAN_GetErrorText(result, 0, strMsg);
```
```csharp
MessageBox(strMsg);
}
else
    MessageBox("PCAN-Dongle (Ch-1) was initialized

....

// All initialized channels are released
//
CAN_Uninitialize(PCAN_NONEBUS);
```

- **See Also**
  - [CAN_Uninitialize](#)
  - [CAN_GetValue](#)
  - [Understanding PCAN-Basic](#)

**Class-method Version:** [Initialize](#)
CAN_InitializeFD

Initializes a FD capable PCAN Channel.

Syntax

```cpp
TPCANStatus __stdcall CAN_InitializeFD(
    TPCANHandle Channel,
    TPCANBitrateFD BitrateFD);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a FD capable PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>BitrateFD</td>
<td>The speed for the communication (FD Bitrate string).</td>
</tr>
</tbody>
</table>

Returns

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

<table>
<thead>
<tr>
<th>PCAN_ERROR_CAUTION:</th>
<th>Indicates that the channel has been initialized but at a different bit rate as the given one.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_I LLHANDLE:</td>
<td>Indicates that the desired PCAN Channel is not valid. Check the list of valid</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PCAN_ERROR_IILLHW:</td>
<td>Indicates that the desired PCAN Channel is not available.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLOPERATION:</td>
<td>Indicates that an action cannot be executed due to the state of the hardware. Possible causes are:</td>
</tr>
<tr>
<td></td>
<td>• The desired PCAN Channel is not FD capable and cannot be initialized using this method.</td>
</tr>
<tr>
<td></td>
<td>• The desired PCAN-Channel is a LAN Channel, which uses a different bit rate than the specified.</td>
</tr>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the desired PCAN channel cannot be connected because it is already in use (PCAN-Basic / PCAN-Light environment).</td>
</tr>
<tr>
<td>PCAN_ERROR_NETINUSE:</td>
<td>Indicates that the desired PCAN-Channel is being used with a different bit rate (PCAN-View).</td>
</tr>
<tr>
<td>PCAN_ERROR_HWINUSE:</td>
<td>Indicates that the desired PCAN-Channel is being used (CanApi connection).</td>
</tr>
<tr>
<td>PCAN_ERROR_NODRIVER:</td>
<td>The driver needed for</td>
</tr>
</tbody>
</table>
Remarks

Note on correspondence of functions:

A Channel that is initialized using CAN_InitializeFD must use
CAN_ReadFD and CAN_WriteFD for communication. Calling
CAN_Read and/or CAN_Write will result in a
PCAN_ERROR_ILLOPERATION error.

As indicated by its name, the CAN_InitializeFD function initiates a FD
capable PCAN Channel, preparing it for communicate within the CAN
bus connected to it. Calls to the API functions will fail if they are used
with a Channel handle, different than PCAN_NONEBUS, that has not
been initialized yet. Each initialized channel should be released when
it is not needed anymore.

Initializing a PCAN Channel means:

- to reserve the Channel for the calling application/process.
- to allocate channel resources, like receive and transmit queues.
- to register/connect the Hardware denoted by the channel handle.
- to check and adapt the bus speed, if the Channel is already
  in use. (Only if the Channel was pre-configured as Bitrate
  Adapting; see: Bitrate-Adapting Parameter).
- to set the channel in Listen-Only mode. (Only if the channel
  was pre-configured as Listen-Only; see: Listen-Only Parameter).
- to open the messages filter for the application/process.
- to set-up the default values of the different parameters (See
  CAN_GetValue).
- to set the Receive Status of the channel. (Pre-configured
  value; see: Receive Status Parameter).

The Initialization process will fail if an application try to initialize a
PCAN-Channel that has been initialized already within the same
process.
Take in consideration that initializing a channel causes a reset of the CAN hardware, when the bus status is other than OK. In this way errors like BUSOFF, BUSWARNING, and BUSPASSIVE, are removed.

**PCAN-LAN Channels**

A PCAN-LAN channel doesn't allow changing the bit rate using PCAN-Basic. In order to connect a PCAN-LAN Channel it is necessary to know the bit rate of the PCAN-Gateway device that is represented by that channel. If the bit rate is not known, the parameter **Bitrate-Adapting** should be used.

**Example**

The following example shows the initialize and uninitialized processes for a FD capable channel (channel 1 of a PCAN-USB Pro FD hardware). In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

**C++:**

```cpp
TPCANStatus result;
char strMsg[256];
TPCANBitrateFD bitrate;

// Defines a FD Bit rate string with nominal and data
// bitrate = "f_clock_mhz=24, nom_brp=1, nom_tseg1=17,
// nom_tseg2=6, nom_sjw=1, data_brp=1, data_tseg1=16,
// data_tseg2=7, data_sjw=1"

// The FD capable Channel (PCAN-USB Pro FD) is initialized
result = CAN_InitializeFD(PCAN_USBBUS1, bitrate);
if(result != PCAN_ERROR_OK) {
    // An error occurred, get a text describing the error
    CAN_GetErrorText(result, 0, strMsg);
}
```
MessageBox(strMsg);
}
else
    MessageBox("PCAN-USB Pro FD (Ch-1) was initialized\n"

See Also

CAN_Uninitialize
CAN_ReadFD
CAN_WriteFD

Class-method Version: InitializeFD
Uninitializes a PCAN Channel.

Syntax

```c++
TPCANStatus __stdcall CAN_Uninitialize(
    TPCANHandle Channel
);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
</tbody>
</table>

Returns

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

- **PCAN_ERROR_INITIALIZE**: Indicates that the given PCAN channel cannot be uninitialized because it was not found in the list of reserved channels of the calling application.

Remarks

A PCAN Channel can be released using one of this possibilities:

- Single-Release: Given a handle of a PCAN Channel initialized before with CAN_Initialize. If the given channel can not be found then an error is returned.
Multiple-Release: Giving the handle value PCAN_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.

**Transmit-queue at uninitialize:** When a PCAN-Basic channel connection is terminated, the underlying hardware’s transmit-queue will not immediately be discarded. PCAN-Basic will wait some time before finalizing, so that the hardware has time to send (or try to send) those unsent messages. When the time is up (amount 500 milliseconds), the rest of the messages in the queue (if any) are discarded.

**Example**

The following example shows the initialize and uninitialized (Single-Release) processes for the PCAN_PCIBUS1 channel. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: To see an example of Multiple-Release, see the CAN_Initialize function.

**C++:**

```cpp
TPCANStatus result;
char strMsg[256];

// The PCI Channel is initialized
//
result = CAN_Initialize(PCAN_PCIBUS1,PCAN_BAUD_500K)
if(result != PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    //
    CAN_GetErrorText(result, 0, strMsg);
    MessageBox(strMsg);
}
```
else
    MessageBox("PCAN-PCI (Ch-1) was initialized");

....

// The PCI Channel is released
//
result = CAN_Uninitialize(PCAN_PCIBUS1);
if (result != PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the error
    //
    CAN_GetErrorText(result, 0, strMsg);
    MessageBox(strMsg);
}
else
    MessageBox("PCAN-PCI (Ch-1) was released");

See Also

CAN_Initialize

Class-method Version: Uninitialize
Resets the receive and transmit queues of a PCAN Channel.

**Syntax**

```cpp
TPCANStatus __stdcall CAN_Reset(
    TPCANHandle Channel
);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see <code>TPCANHandle</code>).</td>
</tr>
</tbody>
</table>

**Returns**

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

- **PCAN_ERROR_INITIALIZE**: Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.

**Remarks**

Calling this function ONLY clear the queues of a Channel. A reset of the CAN controller doesn't take place.

Normally a reset of the CAN Controller is desired when a bus-off occur. In this case an application cannot use the channel to communicate anymore, until the CAN controller is reset. Consider
using the PCAN-Basic parameter **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, but only when no more clients are connected to that channel.

## Example

The following example shows the use of CAN_Reset on the channel PCAN_PCIEBUS1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

**C++:**

```cpp
TPCANStatus result;
char strMsg[256];

// The PCI Channel is reset
//
result = CAN_Reset(PCAN_PCIEBUS1);
if(result != PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the
    //
    CAN_GetErrorText(result, 0, strMsg);
    MessageBox(strMsg);
}
else
    MessageBox("PCAN-PCI (Ch-1) was reset");
```
CAN_Read
CAN_Write
CAN_SetValue

Class-method Version: Reset
Gets the current BUS status of a PCAN Channel.

Syntax

```cpp
TPCANStatus __stdcall CAN_GetStatus(
    TPCANHandle Channel
);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPCANStatus code. The typical return values are:

<table>
<thead>
<tr>
<th>TPCANStatus</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_INITIALIZE</td>
<td>Indicates that the given PCAN Channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSLIGHT</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-light status.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSHEAVY</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-heavy status.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSOFF</td>
<td>Indicates a bus error within the</td>
</tr>
</tbody>
</table>


given PCAN Channel. The hardware is in bus-off status.

| **PCAN_ERROR_OK:** | Indicates that the status of the given PCAN Channel is OK. |

## 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 CAN_GetStatus on the channel PCAN_PCIEBUS1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

**C++:**

```cpp
TPCANStatus result;
char strMsg[256];

// Check the status of the PCI Channel
//
result = CAN_GetStatus(PCAN_PCIEBUS1);
switch(result)
{
    case PCAN_ERROR_BUSLIGHT:
        MessageBox("PCAN-PCI (Ch-1): Handling a BU");
```
break;
case PCAN_ERROR_BUSHEAVY:
    MessageBox("PCAN-PCI (Ch-1): Handling a BUS-HEAVY status..."
break;
case PCAN_ERROR_BUSOFF:
    MessageBox("PCAN-PCI (Ch-1): Handling a BUS-OFF status..."
break;
case PCAN_ERROR_OK:
    MessageBox("PCAN-PCI (Ch-1): Status is OK"
break;
default:
    // An error occurred, get a text describing
    // CAN_GetErrorText(result, 0, strMsg);
    MessageBox(strMsg);
    break;
}

See Also

Parameter Value Definitions
TPCANParameter

Class-method Version: GetStatus
Can_Read

Reads a CAN message from the receive queue of a PCAN Channel.

Syntax

C++

```cpp
TPCANStatus __stdcall CAN_Read(
    TPCANHandle Channel,
    TPCANMsg* MessageBuffer,
    TPCANTimestamp* TimestampBuffer
);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>MessageBuffer</td>
<td>A TPCANMsg buffer to store the CAN message.</td>
</tr>
<tr>
<td>TimestampBuffer</td>
<td>A TPCANTimestamp buffer to get the reception time of the message.</td>
</tr>
</tbody>
</table>

Returns

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

- **PCAN_ERROR_IILLPARAMVAL**: Indicates that the parameters passed to the function are invalid. Check the value of the
MessageBuffer; it should point to a `TPCANMsg` structure.

**PCAN_ERROR_INITIALIZE:** Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.

**PCAN_ERROR_BUSLIGHT:** Indicates a bus error within the given PCAN Channel. The hardware is in bus-light status.

**PCAN_ERROR_BUSHEAVY:** Indicates a bus error within the given PCAN Channel. The hardware is in bus-heavy status.

**PCAN_ERROR_BUSOFF:** Indicates a bus error within the given PCAN Channel. The hardware is in bus-off status.

**PCAN_ERROR_QRCVEMPTY:** Indicates that the receive queue of the Channel is empty.

### Remarks

Specifying the value of NULL for the parameter TimestampBuffer causes reading a message without timestamp, when the reception time is not desired. An "Illegal Parameter Value" error will be returned when the MessageBuffer is wrong or the TimestampBuffer contains a value different than NULL and provokes an internal error, eg. accessing its memory.

The use of CAN_Read and **CAN_ReadFD** are mutually exclusive. The PCAN Channel passed to this function must be initialized using **CAN_Initialize** (class-method: `Initialize`). Otherwise the error **PCAN_ERROR_ILLOPERATION** is returned.

The CAN_Read function returns received messages or status
messages from the receive queue. It is important to call CAN_Read repeatedly until the queue is empty. In case there are no more messages in queue, the value PCAN_ERROR_QRCVEMPTY is returned. The error code PCAN_ERROR_QRCVEMPTY is also returned if the reception of messages is disabled. See Receive Status Parameter for more information.

The receive queue can contain up to 32767 messages.

There are two possibilities for reading messages from the receive queue of a Channel:

*Time-Triggered Reading:* Consists in periodically calls to the CAN_Read function. Typically, an application start a timer that every 50 or 100 milliseconds check for messages, calling the CAN_Read function in a loop until the value of PCAN_ERROR_QRCVEMPTY or another error condition is reached.

*Event-Triggered Reading:* Consists in reacting to a notification sent by the PCAN driver to a registered application, when a message is received and inserted in its receive queue. See Using Events to obtain more information about reading with events.

**About bus errors / Status messages**

If a bus-off error occur, an application cannot use the channel to communicate anymore, until the CAN controller is reset. With PCAN-Basic it is not possible to reset the CAN controller through a function directly. 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 BUSOFF, BUSHEAVY, and BUSLIGHT, is to uninitialize and initialise again the channel used. This causes a hardware reset.

The message type (see TPCANMessageType) of a CAN message indicates if the message is a 11-bit, 29-bit, RTR, Error, or Status message. This value should be checked every time a message has been read successfully.
If the bit \texttt{PCAN\_MESSAGE\_ERRFRAME} is set in the \texttt{TPCANMsg.MSGTYPE} field, the message is an Error frame (see Error Frames).

If the bit \texttt{PCAN\_MESSAGE\_STATUS} is set in the \texttt{TPCANMsg.MSGTYPE} field, the message is a Status message. The ID and LEN fields do not contain valid data. The first 4 data bytes of the message contain the Error Code. The MSB of the Error Code is in data byte 0, the LSB is in data byte 3. If a status message was read the return value of \texttt{CAN\_Read} is also the error code.

Examples:

<table>
<thead>
<tr>
<th>Data0</th>
<th>Data1</th>
<th>Data2</th>
<th>Data3</th>
<th>Error</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>02h</td>
<td>PCAN_ERROR_OVERRUN</td>
<td>0002h</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>04h</td>
<td>PCAN_ERROR_BUSLIGHT</td>
<td>0004h</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>08h</td>
<td>PCAN_ERROR_BUSHEAVY</td>
<td>0008h</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>10h</td>
<td>PCAN_ERROR_BUSOFF</td>
<td>0010h</td>
</tr>
</tbody>
</table>
Example

The following example shows the use of CAN_Read on the channel PCAN_USB1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: It is assumed that the channel was already initialized and that the following code is an OnTimer event handler function.

C++:

```cpp
TPCANMsg msg;
TPCANTimestamp timestamp;
TPCANStatus result;
char strMsg[256];

do
{
    // Check the receive queue for new messages
    //
    result = CAN_Read(PCAN_USB1,&msg,&timestamp;
    if(result != PCAN_ERROR_QRCVEMPTY)
    {
        // Process the received message
        //
        MessageBox("A message was received");
        ProcessMessage(msg)
    }
    else
    {
        // An error occurred, get a text describing
        // and handle the error
        //
        CAN_GetErrorText(result, 0, strMsg);
        MessageBox(strMsg);
        // Here can be decided if the loop has to
        // status is bus-off)
    }
}```
//
HandleReadError(result);
//
}
// Try to read a message from the receive queue of
// until the queue is empty
//
}while((result & PCAN_ERROR_QRCVEMPTY) != PCAN_ERR

See Also

CAN_Write
Using Events
Error Frames

Class-method Version: Read
CAN_ReadFD

Reads a CAN message from the receive queue of a FD capable PCAN Channel.

Syntax

### C++

```cpp
TPCANStatus __stdcall CAN_ReadFD(
    TPCANHandle Channel,
    TPCANMsgFD* MessageBuffer,
    TPCANTimestampFD *TimestampBuffer);
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a FD capable PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>MessageBuffer</td>
<td>A TPCANMsgFD buffer to store the CAN message.</td>
</tr>
<tr>
<td>TimestampBuffer</td>
<td>A TPCANTimestampFD buffer to get the reception time of the message.</td>
</tr>
</tbody>
</table>

### Returns

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

- **PCAN_ERROR_IILLPARAMVAL:** Indicates that the parameters passed to the function are
invalid. Check the value of the MessageBuffer; it should point to a **TPCANMsgFD** structure.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_ILLOPERATION:</td>
<td>Indicates that the PCAN Channel passed to the function was not initialized using <strong>CAN_InitializeFD</strong> (class-method: <strong>InitializeFD</strong>).</td>
</tr>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSWARNING:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-warning status.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSPASSIVE:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-passive status.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSOFF:</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-off status.</td>
</tr>
<tr>
<td>PCAN_ERROR_QRCVEMPTY:</td>
<td>Indicates that the receive queue of the Channel is empty.</td>
</tr>
</tbody>
</table>

**Remarks**

Specifying the value of NULL for the parameter TimetampBuffer causes reading a message without timestamp, when the reception
time is not desired. An "Illegal Parameter Value" error will be returned when the MessageBuffer is wrong or the TimestampBuffer contains a value different than NULL and provokes an internal error, eg. accessing its memory.

The use of CAN_Read and CAN_ReadFD are mutually exclusive. The PCAN Channel passed to this function must be initialized using CAN_InitializeFD (class-method: InitializeFD). Otherwise the error PCAN_ERROR_ILLOPERATION is returned.

The CAN_ReadFD function returns received messages or status messages from the receive queue. It is important to call CAN_ReadFD repeatedly until the queue is empty. In case there are no more messages in queue, the value PCAN_ERROR_QRCVEMPTY is returned. The error code PCAN_ERROR_QRCVEMPTY is also returned if the reception of messages is disabled. See Receive Status Parameter for more information.

The receive queue can contain up to 32767 messages.

There are two possibilities for reading messages from the receive queue of a Channel:

**Time-Triggered Reading:** Consists in periodically calls to the CAN_ReadFD function. Typically, an application start a timer that every 50 or 100 milliseconds check for messages, calling the CAN_ReadFD function in a loop until the value of PCAN_ERROR_QRCVEMPTY or another error condition is reached.

**Event-Triggered Reading:** Consists in reacting to a notification sent by the PCAN driver to a registered application, when a message is received and inserted in its receive queue. See Using Events to obtain more information about reading with events.

**About bus errors / Status messages**

If a bus-off error occur, an application cannot use the channel to communicate anymore, until the CAN controller is reset. With PCAN-Basic it is not possible to reset the CAN controller through a function directly. 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 BUSOFF, BUSWARNING, and BUSPASSIVE, is to **uninitialize** and **initialize** again the channel used. This causes a hardware reset.

The message type (see **TPCANMessageType**) of a CAN message indicates if the message is a 11-bit, 29-bit, FD, RTR, Error, or Status message. This value should be checked every time a message has been read successfully.

If the bit **PCAN_MESSAGE_ERRFRAME** is set in the **TPCANMsg.MSGTYPE** field, the message is an Error frame (see **Error Frames**).

If the bit **PCAN_MESSAGE_STATUS** is set in the **TPCANMsg.MSGTYPE** field, the message is a Status message. The ID and DLC fields do not contain valid data. The first 4 data bytes of the message contain the Error Code. The MSB of the Error Code is in data byte 0, the LSB is in data byte 3. If a status message was read the return value of **CAN_ReadFD** is also the error code.

Examples:

<table>
<thead>
<tr>
<th>Data0</th>
<th>Data1</th>
<th>Data2</th>
<th>Data3</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>02h</td>
<td>PCAN_ERROR_OVERRUN</td>
</tr>
<tr>
<td>00h</td>
<td>00h</td>
<td>00h</td>
<td>08h</td>
<td>PCAN_ERROR_BUSWARNING</td>
</tr>
</tbody>
</table>


Example

The following example shows the use of CAN_ReadFD on the channel PCAN_USBBUS1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: It is assumed that the channel was already initialized using CAN_InitializeFD and that the following code is an OnTimer event handler function.

C++:

```cpp
tpcanmsgfd msg;
tpcantimestampfd timestamp;
tpcanstatus result;
char strmsg[256];

do{
    // Check the receive queue for new messages
    //
    result = CAN_ReadFD(PCAN_USBBUS1,&msg,&timestamp);
    if(result != PCAN_ERROR_QRCVEMPTY)
    {
        // Process the received message
```
//
MessageBox("A message was received");
ProcessMessage(msg)
}
else
{
    // An error occurred, get a text describing
    // and handle the error
    //
    CAN_GetErrorText(result, 0, strMsg);
    MessageBox(strMsg);
    // Here can be decided if the loop has to
    // status is bus-off)
    //
    HandleReadError(result);
}
// Try to read a message from the receive queue of
// until the queue is empty
//
}while((result & PCAN_ERROR_QRCVEMPTY) != PCAN_ERR

See Also

CAN_InitializeFD
CAN_WriteFD
Using Events
Error Frames

Class-method Version: ReadFD
CAN_Write

Transmits a CAN message.

Syntax

```cpp
TPCANStatus __stdcall CAN_Write(
    TPCANHandle Channel,
    TPCANMsg* MessageBuffer
);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>MessageBuffer</td>
<td>A TPCANMsg buffer containing the CAN message to be sent.</td>
</tr>
</tbody>
</table>

Returns

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

<table>
<thead>
<tr>
<th>PCAN_ERROR_IILLPARAMVAL:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicates that the parameters passed to the function are invalid. Check the value of the MessageBuffer; it should point to a TPCANMsg structure.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PCAN_ERROR_INITIALIZE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of</td>
</tr>
</tbody>
</table>
### Remarks

If a bus-off error occur, an application cannot use the channel to communicate anymore, until the CAN controller is reset. With PCAN-Basic it is not possible to reset the CAN controller through a function directly. 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 BUSOFF, BUSHEAVY, and BUSLIGTH, is to `uninitialize` and `initialise` again the channel used. This causes a hardware reset, but only when no more clients are connected to that channel.

### Example

The following example shows the use of CAN_Write on the channel PCAN_USBUSB1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

**C++:**

```cpp
TPCANMsg msg;
TPCANStatus result;
char strMsg[256];
```
// A CAN message is configured
//
msg.ID = 0x100;
msg.MSGTYPE = PCAN_MESSAGE_STANDARD;
msg.LEN = 3;
msg.DATA[0] = 1;
msg.DATA[1] = 2;
msg.DATA[2] = 3;

// The message is sent using the PCAN-USB Channel
//
result = CAN_Write(PCAN_USBBUS1, &msg);
if(result != PCAN_ERROR_OK)
{
    // An error occurred, get a text describing th
    //
    CAN_GetErrorText(result, 0, strMsg);
    MessageBox(strMsg);
}
else
    MessageBox("Message sent successfully");

See Also

CAN_Read
CAN_SetValue

Class-method Version: Write
CAN_WriteFD

Transmits a CAN message using a connected FD capable PCAN Channel.

Syntax

```
C++
TPCANStatus __stdcall CAN_WriteFD(
    TPCANHandle Channel,
    TPCANMsgFD * MessageBuffer);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a FD capable PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>MessageBuffer</td>
<td>A TPCANMsgFD buffer containing the CAN message to be sent.</td>
</tr>
</tbody>
</table>

Returns

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

<p>| PCAN_ERROR_IILLPARAMVAL: | Indicates that the parameters passed to the function are invalid. Check the value of the MessageBuffer; it should point to a TPCANMsgFD structure. |</p>
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCAN_ERROR_ILLOPERATION</td>
<td>Indicates that the PCAN Channel passed to the function was not initialized using <em>CAN_InitializeFD</em> (class-method: <em>InitializeFD</em>).</td>
</tr>
<tr>
<td>PCAN_ERROR_INITIALIZE</td>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td>PCAN_ERROR_BUSOFF</td>
<td>Indicates a bus error within the given PCAN Channel. The hardware is in bus-off status.</td>
</tr>
<tr>
<td>PCAN_ERROR_QXMTFULL</td>
<td>Indicates that the transmit queue of the Channel is full.</td>
</tr>
</tbody>
</table>

**Remarks**

The use of *CAN_Write* and *CAN_WriteFD* are mutually exclusive. The PCAN Channel passed to this function must be initialized using *CAN_InitializeFD* (class-method: *InitializeFD*). Otherwise the error *PCAN_ERROR_ILLOPERATION* is returned.

If a bus-off error occur, an application cannot use the channel to communicate anymore, until the CAN controller is reset. With PCAN-Basic it is not possible to reset the CAN controller through a function directly. 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 BUSOFF, BUSWARNING, and BUSPASSIVE, is to *uninitialize* and *initialise* again the channel used. This causes a hardware reset.

**Example**
The following example shows the use of CAN_WriteFD on the channel PCAN_USBBUS1. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

C++:

```cpp
TPCANMsgFD msg;
TPCANStatus result;
char strMsg[256];

// A CAN FD message is configured
//
msg.ID = 0x100;
msg.MSGTYPE = PCAN_MESSAGE_STANDARD | PCAN_MESSAGE_FD
// DLC 9 means 12 data bytes
//
msg.DLC = 9;
for(int i=0; i < 12; i++)
    msg.DATA[i] = i;

// The message is sent using the PCAN-USB Channel
//
result = CAN_WriteFD(PCAN_USBBUS1, &msg);
if(result != PCAN_ERROR_OK)
{
    // An error occurred, get a text describing th
    //
    CAN_GetErrorText(result, 0, strMsg);
    MessageBox(strMsg);
}
else
    MessageBox("Message sent successfully");
```
See Also

CAN_InitializeFD
CAN_ReadFD

Class-method Version: WriteFD
**PCAN-Basic Documentation**

**CAN_GetValue**

Retrieves information from a PCAN Channel.

**Syntax**

```
#include <PCANBasic.h>

TPCANStatus __stdcall CAN_GetValue(
    TPCANHandle Channel,
    TPCANParameter Parameter,
    void* Buffer,
    WORD BufferLength
);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see <a href="#">TPCANHandle</a>).</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the information to be retrieved (see <a href="#">TPCANParameter</a>).</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 [TPCANStatus](#) code. PCAN_ERROR_OK is returned on success. The typical errors in case of failure are:
**PCAN_ERROR_ILLPARAMVAL:** Indicates that the parameters passed to the function are invalid. Check the parameter 'Buffer'; it should point to a valid data container for the requested value.

**PCAN_ERROR_INITIALIZE:** Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.

**PCAN_ERROR_ILLPARAMTYPE:** Indicates that the requested information is not available for the given PCAN Channel. Check the value of 'Parameter'; some values are not available for all PCAN-Channels or cannot be read.

**Remarks**

Use the function CAN_GetValue to get information about PCAN environment as parameters like the Message Filter and values like the availability of a PCAN-Channel. Take in account that not all parameters are supported for all PCAN-Channels. The access's type of the parameters can also be different.

More information about the parameters and values that can be read can be found in [Parameter Value Definitions](#).

**Example**

The following example shows the use of CAN_GetValue on the channel PCAN_USBBUS1 to check if the Message Filter is fully opened. In case of failure, the returned code will be translated to a
text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user. Note: It is assumed that the channel was already initialized.

C++:

```c++
int iBuffer;
TPCANStatus result;
char strMsg[256];

// The status of the message filter of the PCAN-US
// result = CAN_GetValue(PCAN_USBBUS1, PCAN_MESSAGE_FILTER,
if(result != PCAN_ERROR_OK)
{
   // An error occurred, get a text describing the
   //
   CAN_GetErrorText(result, 0, strMsg);
   MessageBox(strMsg);
}
else
{
   // A text is shown giving information about the
   //
   switch(result != PCAN_ERROR_OK)
   {
      case PCAN_FILTER_OPEN:
         MessageBox("The message filter for the PCAN-USB, channel 1, is completely opened.");
         break;
      case PCAN_FILTER_CLOSE:
         MessageBox("The message filter for the PCAN-USB, channel 1, is closed.");
         break;
      case PCAN_FILTER_CUSTOM:
         MessageBox("The message filter for the PCAN-USB, channel 1, is custom configured.");
         break;
   }
}
```
See Also

CAN_SetValue
TPCANParameter
Parameter Value Definitions

Class-method Version: GetValue
Sets a configuration or information value within a PCAN Channel.

Syntax

```cpp
TPCANStatus __stdcall CAN_SetValue(
    TPCANHandle Channel,
    TPCANParameter Parameter,
    void* Buffer,
    WORD BufferLength
);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>Parameter</td>
<td>The code of the value to be set (see TPCANParameter).</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 TPCANStatus code. PCAN_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>PCAN_ERROR_ILLPARAMVAL:</td>
<td>Indicates that the parameters passed to the function are invalid. Check the parameter 'Buffer'; it should point to a valid data container for the requested value.</td>
</tr>
<tr>
<td>PCAN_ERROR_CAUTION:</td>
<td>The configuration of a parameter failed due to a no more existing channel. The parameter has been reset on all existing channels.</td>
</tr>
<tr>
<td>PCAN_ERROR_INITIALIZE:</td>
<td>Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLPARAMTYPE:</td>
<td>Indicates that the requested information is not available for the given PCAN Channel. Check the value of 'Parameter'; some values are not available for all PCAN-Channels or cannot be set.</td>
</tr>
<tr>
<td>PCAN_ERROR_ILLOPERATION:</td>
<td>An underlying process that is generated by a call to this function with the current parameters, is temporarily not allowed. The configuration in relation to the used TPCANParameter must be checked.</td>
</tr>
</tbody>
</table>

Remarks
Use the function CAN_SetValue to set configuration information or environment values of a PCAN Channel as parameters like the Message Filter and values like a custom entry in the log file of PCAN-Basic. Take in account that not all parameters are supported for all PCAN-Channels. The access's type of the parameters can also be different.

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 CAN_SetValue on the channel PCAN_USBBUS1 to insert a text into the PCAN-Basic log file. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

Note: It is not needed to have an initialized PCAN channel for using the Log functionality.

**C++:**

```c++
TPCANStatus result;
char strMsg[256];

// Sets a text to be included in the Log file of the PCAN Channel
strcpy(strMsg, "This is a custom text from an application.

// Inserts the given text into the Log file of the PCAN Channel.
// Note: If the Log functionality is disabled, this call will automatically activate the log process.
result = CAN_SetValue(PCAN_NONEBUS, PCAN_LOG_TEXT, if(result != PCAN_ERROR_OK)

    // An error occurred, get a text describing the error
    
```
See Also

- **CAN_GetValue**
- **TPCANParameter**
- **Parameter Value Definitions**

Class-method Version: **SetValue**
CAN_FilterMessages

Configures the reception filter.

Syntax

```cpp
TPCANStatus __stdcall CAN_FilterMessages(
    TPCANHandle Channel,
    DWORD FromID,
    DWORD ToID,
    TPCANMode Mode
);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>The handle of a PCAN Channel (see TPCANHandle).</td>
</tr>
<tr>
<td>FromID</td>
<td>The lowest CAN ID wanted to be received.</td>
</tr>
<tr>
<td>ToID</td>
<td>The highest CAN ID wanted to be received.</td>
</tr>
<tr>
<td>Mode</td>
<td>The type of the filter being set (see TPCANType).</td>
</tr>
</tbody>
</table>

Returns

The return value is a TPCANStatus code. PCAN_ERROR_OK is returned on success. The typical errors in case of failure are:
PCAN_ERROR_INITIALIZE: Indicates that the given PCAN channel was not found in the list of initialized channels of the calling application.

Remarks

Note that after a PCAN Channel is initialized, the status of its filter is fully opened. According with the current filter status, calling this function causes the following behavior:

- Filter status is PCAN_FILTER_OPEN: The filter is automatically closed and then configured with the given range of IDs passed to this function [FromID, ToID].
- Filter status is PCAN_FILTER_CLOSE: The filter is set to the given range of IDs passed to this function [FromID, ToID].
- Filter status is PCAN_FILTER_CUSTOM: The filter is expanded with the given range of IDs [FromID, ToID]. If a smaller or different range is required than a range that has been configured before, the filter has to be closed first before calling the CAN_FilterMessages function. To do this use the function CAN_SetValue.

The parameter 'Mode' indicates which kind of ID is being used to register the new filter range. There are two possible values, Standard (11-bit identifier) or Extended (29-bit identifier). Standard frames are using the bit positions 28 to 18 of the Acceptance Mask/Code registers in the SJA1000 CAN controller. Drivers for 82C200 CAN controllers have to shift the bits down to positions 10 to 0.

Take in account that configuring the message filter cause the CAN controller to enter the Reset state. This will affect other applications that communicate with the same PCAN hardware.

Notes:

1. There is only one filter for standard and extended CAN messages. It seems that the ID from a standard message uses the most significant 11 bits (bit 18 to 28) of the 29 bits. i.e. the standard ID 400h is also received by indicating an extended ID
10000000h. For this reason it is not recommended to mix standard and extended filters, since it can increase the risk of receiving unwanted messages.

2. Multiple calls of **CAN_FilterMessages** expand the reception filter.

3. It is not guaranteed that an application only receives CAN messages in the range of **FromID** to **ToID**. This is caused by the operating principle of the SJA1000’s acceptance filter. See also Philips Data Sheet "SJA1000 Stand-alone CAN-controller".

**Example**

The following example shows the use of CAN_FilterMessages on the channel PCAN_USBUSBUS1 to receive a custom range of IDs. In case of failure, the returned code will be translated to a text (according with the operating system language) in English, German, Italian, French or Spanish, and it will be shown to the user.

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

**C++:**

```cpp
TPCANStatus result;
char strMsg[256];
DWORD iBuffer;

// The message filter is closed first to ensure that
// iBuffer = PCAN_FILTER_CLOSE;
result = CAN_SetValue(PCAN_USBUSBUS1, PCAN_MESSAGE_FILTER, &iBuffer);

if(result != PCAN_ERROR_OK)
{
    // An error occurred, get a text describing the
    // CAN_GetErrorText(result, 0, strMsg);
    MessageBox(strMsg);
}
else
```
The message filter is configured to receive

result = CAN_FilterMessages(PCAN_USBBUS1, 2, 5)

if(result != PCAN_ERROR_OK)
{
    // An error occurred, get a text describing it
    //
    CAN_GetErrorText(result, 0, strMsg);
    MessageBox(strMsg);
}
else
    MessageBox("Filter successfully configured")

See Also

CAN_SetValue

Class-method Version: FilterMessages
Returns a descriptive text for an error code.

Syntax

```cpp
TPCANStatus __stdcall CAN_GetErrorText(
    TPCANStatus Error,
    WORD Language,
    LPSTR Buffer
);
```

### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>A <code>TPCANStatus</code> error code.</td>
</tr>
<tr>
<td>Language</td>
<td>Indicates a &quot;Primary language ID&quot;.</td>
</tr>
<tr>
<td>Buffer</td>
<td>A buffer for a null-terminated char array.</td>
</tr>
</tbody>
</table>

### Returns

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

- **PCAN_ERROR_IILPARAMVAL**: Indicates that the parameters passed to the function are invalid. Check the parameter 'Buffer'; it should point to a char array, big enough to...
Remarks

The "Primary language IDs" are codes used by Windows OS from Microsoft, to identify a human language. The PCAN-Basic API currently support the following languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Primary Language ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neutral (English)</td>
<td>00h (0)</td>
</tr>
<tr>
<td>English</td>
<td>09h (9)</td>
</tr>
<tr>
<td>German</td>
<td>07h (7)</td>
</tr>
<tr>
<td>French</td>
<td>0Ch (12)</td>
</tr>
<tr>
<td>Italian</td>
<td>10h (16)</td>
</tr>
<tr>
<td>Spanish</td>
<td>0Ah (10)</td>
</tr>
</tbody>
</table>

Note: If the buffer is too small for the resulting text, the error PCAN_ERROR_ILLPARAMVAL is returned. Even when only short texts are being currently returned, a text within this function can have a maximum of 255 characters. For this reason it is recommended to use a buffer with a length of at least 256 bytes.

Example

The following example shows the use of CAN_GetErrorText to get the description of an error. The language of the description's text will be the same used by the operating system.

C++:

```cpp
TPCANStatus result;
char strMsg[256];

// Gets the description text for PCAN_ERROR_INITIATION
```
/
result = CAN_GetErrorText(PCAN_ERROR_INITIALIZE, \0
if(result != PCAN_ERROR_OK)
  // An error occurred, get a text describing the error
  MessageBox("Error when recovering Error-Code's description");
else
  MessageBox(strMsg);

See Also

- Primary Language ID

Class-method Version: GetErrorText
The PCAN-Basic API defines the following values:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PCAN Handle Definitions</strong></td>
<td>Defines the handles for the different PCAN Channels.</td>
</tr>
<tr>
<td><strong>Parameter Value Definitions</strong></td>
<td>Defines the possible values for setting and getting PCAN's environment information with the functions CAN_SetValue and CAN_GetValue.</td>
</tr>
<tr>
<td><strong>FD Bit rate Parameter Definitions</strong></td>
<td>Defines the different configuration parameters used to create a Flexible Data rate string for FD capable PCAN-Channels initialization.</td>
</tr>
</tbody>
</table>
PCAN Handle Definitions

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

**Default** handle value:

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TPCANHandle</code></td>
<td>PCAN_NONEBUS</td>
<td>0</td>
<td>Undefined/default value for a PCAN bus.</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>TPCANHandle</code></td>
<td>PCAN_ISABUS1</td>
<td>0x21</td>
<td>PCAN-ISA interface, channel 1.</td>
</tr>
<tr>
<td><code>TPCANHandle</code></td>
<td>PCAN_ISABUS2</td>
<td>0x22</td>
<td>PCAN-ISA interface, channel 2.</td>
</tr>
<tr>
<td><code>TPCANHandle</code></td>
<td>PCAN_ISABUS3</td>
<td>0x23</td>
<td>PCAN-ISA interface, channel 3.</td>
</tr>
<tr>
<td><code>TPCANHandle</code></td>
<td>PCAN_ISABUS4</td>
<td>0x24</td>
<td>PCAN-ISA interface, channel 4.</td>
</tr>
<tr>
<td><code>TPCANHandle</code></td>
<td>PCAN_ISABUS5</td>
<td>0x25</td>
<td>PCAN-ISA interface, channel 5.</td>
</tr>
<tr>
<td>Type</td>
<td>Constant</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
<td>-------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_ISABUS6</td>
<td>0x26</td>
<td>PCAN-ISA interface, channel 6.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_ISABUS7</td>
<td>0x27</td>
<td>PCAN-ISA interface, channel 7.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_ISABUS8</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>TPCANHandle</td>
<td>PCAN_DNGBUS1</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>TPCANHandle</td>
<td>PCAN_PCIBUS1</td>
<td>0x41</td>
<td>PCAN-PCI interface, channel 1.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS2</td>
<td>0x42</td>
<td>PCAN-PCI interface, channel 2.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS3</td>
<td>0x43</td>
<td>PCAN-PCI interface, channel 3.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUSx</td>
<td>0xN</td>
<td>PCAN-PCI interface, channel N.</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS4</td>
<td>0x44</td>
<td>PCAN-PCI interface, channel 4.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS5</td>
<td>0x45</td>
<td>PCAN-PCI interface, channel 5.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS6</td>
<td>0x46</td>
<td>PCAN-PCI interface, channel 6.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS7</td>
<td>0x47</td>
<td>PCAN-PCI interface, channel 7.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS8</td>
<td>0x48</td>
<td>PCAN-PCI interface, channel 8.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS9</td>
<td>0x409</td>
<td>PCAN-PCI interface, channel 9.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS10</td>
<td>0x40A</td>
<td>PCAN-PCI interface, channel 10.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS11</td>
<td>0x40B</td>
<td>PCAN-PCI interface, channel 11.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS12</td>
<td>0x40C</td>
<td>PCAN-PCI interface, channel 12.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS13</td>
<td>0x40D</td>
<td>PCAN-PCI interface, channel 13.</td>
</tr>
<tr>
<td>Type</td>
<td>Constant</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------</td>
<td>-------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS14</td>
<td>0x40E</td>
<td>PCAN-PCI interface, channel 14.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS15</td>
<td>0x40F</td>
<td>PCAN-PCI interface, channel 15.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCIBUS16</td>
<td>0x410</td>
<td>PCAN-PCI interface, channel 16.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCANHandle</td>
<td>PCAN_USBBUS1</td>
<td>0x51</td>
<td>PCAN-USB interface, channel 1.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_USBBUS2</td>
<td>0x52</td>
<td>PCAN-USB interface, channel 2.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_USBBUS3</td>
<td>0x53</td>
<td>PCAN-USB interface, channel 3.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_USBBUS4</td>
<td>0x54</td>
<td>PCAN-USB interface, channel 4.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_USBBUS5</td>
<td>0x55</td>
<td>PCAN-USB interface, channel 5.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_USBBUS6</td>
<td>0x56</td>
<td>PCAN-USB interface, channel 6.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_USB</td>
<td>Interface, Channel</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>--------------------</td>
<td></td>
</tr>
<tr>
<td>PCAN_USBBUS7</td>
<td>0x57</td>
<td>PCAN-USB interface, channel 7.</td>
<td></td>
</tr>
<tr>
<td>PCAN_USBBUS8</td>
<td>0x58</td>
<td>PCAN-USB interface, channel 8.</td>
<td></td>
</tr>
<tr>
<td>PCAN_USBBUS9</td>
<td>0x509</td>
<td>PCAN-USB interface, channel 9.</td>
<td></td>
</tr>
<tr>
<td>PCAN_USBBUS10</td>
<td>0x50A</td>
<td>PCAN-USB interface, channel 10.</td>
<td></td>
</tr>
<tr>
<td>PCAN_USBBUS11</td>
<td>0x50B</td>
<td>PCAN-USB interface, channel 11.</td>
<td></td>
</tr>
<tr>
<td>PCAN_USBBUS12</td>
<td>0x50C</td>
<td>PCAN-USB interface, channel 12.</td>
<td></td>
</tr>
<tr>
<td>PCAN_USBBUS13</td>
<td>0x50D</td>
<td>PCAN-USB interface, channel 13.</td>
<td></td>
</tr>
<tr>
<td>PCAN_USBBUS14</td>
<td>0x50E</td>
<td>PCAN-USB interface, channel 14.</td>
<td></td>
</tr>
<tr>
<td>PCAN_USBBUS15</td>
<td>0x50F</td>
<td>PCAN-USB interface, channel 15.</td>
<td></td>
</tr>
<tr>
<td>PCAN_USBBUS16</td>
<td>0x510</td>
<td>PCAN-USB interface, channel 16.</td>
<td></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>TPCANHandle</td>
<td>PCAN_PCCBUS1</td>
<td>0x61</td>
<td>PCAN-PC Card interface, channel 1.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_PCCBUS2</td>
<td>0x62</td>
<td>PCAN-PC Card interface, channel 2.</td>
</tr>
</tbody>
</table>

Handles for the **LAN** Bus:

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCANHandle</td>
<td>PCAN_LANBUS1</td>
<td>0x801</td>
<td>PCAN-LAN interface, channel 1.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_LANBUS2</td>
<td>0x802</td>
<td>PCAN-LAN interface, channel 2.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_LANBUS3</td>
<td>0x803</td>
<td>PCAN-LAN interface, channel 3.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_LANBUS4</td>
<td>0x804</td>
<td>PCAN-LAN interface, channel 4.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_LANBUS5</td>
<td>0x805</td>
<td>PCAN-LAN interface, channel 5.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_LANBUS6</td>
<td>0x806</td>
<td>PCAN-LAN interface, channel 6.</td>
</tr>
<tr>
<td>TPCANHandle</td>
<td>PCAN_LANBUS</td>
<td>0x80</td>
<td>PCAN-LAN interface, channel</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
<td>7.</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>8.</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
<td>9.</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td>10.</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td>11.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
<td>12.</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td></td>
<td>13.</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td></td>
<td>14.</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td></td>
<td>15.</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>16.</td>
</tr>
</tbody>
</table>
Remarks

The PCAN_NONEBUS is a value used as default channel value. It is used for general purposes as using and configuring the Log capabilities of the PCAN-Basic API. It can also be used to remove all channel connections made by an application.

These definitions are constants values in an object oriented environment (Delphi, .NET Framework) and declared as defines in C++ (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 PCAN_ISABUS3 without having registered PCAN_ISA1 and PCAN_ISA2. It is a decision of each user, how to associate a PCAN-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 mean, that having four PCAN-USB connected to a computer will let the user to connect the channels PCAN_USBBUS1 to PCAN_USBBUS4. The association of each channel with a hardware is managed internally using the handle of a hardware.

Python:

The definitions of these values have the following form: 

TPCANHandle(handle) where handle is the value contained in the column with the same name. e.g. PCAN_PCCBUS1 is defined as TPCANHandle(0x61).

See Also

Parameter Value Definitions
Parameter Value Definitions

Defines the possible values for setting and getting PCAN's environment information with the functions `CAN_SetValue` and `CAN_GetValue`.

**Activation values:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>PCAN_PARAMETER_OFF</td>
<td>0</td>
<td>The PCAN parameter is not set (inactive).</td>
</tr>
<tr>
<td>Int32</td>
<td>PCAN_PARAMETER_ON</td>
<td>1</td>
<td>The PCAN parameter is set (active).</td>
</tr>
</tbody>
</table>

**Filter values:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>PCAN_FILTER_CLOSE</td>
<td>0</td>
<td>The PCAN filter is closed. No messages will be received.</td>
</tr>
<tr>
<td>Int32</td>
<td>PCAN_FILTER_OPEN</td>
<td>1</td>
<td>The PCAN filter is fully opened. All messages will be received.</td>
</tr>
<tr>
<td>Int32</td>
<td>PCAN_FILTER_CUSTOM</td>
<td>2</td>
<td>The PCAN filter is custom configured. Only registered messages will be received.</td>
</tr>
</tbody>
</table>
messages will be received.

Channel Availability values:

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>PCAN_CHANNEL_UNAVAILABLE</td>
<td>0</td>
</tr>
<tr>
<td>Int32</td>
<td>PCAN_CHANNELAVAILABLE</td>
<td>1</td>
</tr>
<tr>
<td>Int32</td>
<td>PCAN_CHANNEL_OCCUPIED</td>
<td>2</td>
</tr>
<tr>
<td>Int32</td>
<td>PCAN_CHANNEL_PCANVIEW</td>
<td>PCAN_CHANNEL_AVAILABLE or PCAN_CHANNEL_OCCUPIED</td>
</tr>
</tbody>
</table>
Log-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>LOG_FUNCTION_DEFAULT</td>
<td>0x00</td>
<td>Logs system exceptions and errors. Custom log texts are also included.</td>
</tr>
<tr>
<td>Int32</td>
<td>LOG_FUNCTION_ENTRY</td>
<td>0x01</td>
<td>Logs the entries to the PCAN-Basic API functions.</td>
</tr>
<tr>
<td>Int32</td>
<td>LOG_FUNCTION_PARAMETERS</td>
<td>0x02</td>
<td>Logs the parameters passed to the PCAN-Basic API functions.</td>
</tr>
<tr>
<td>Int32</td>
<td>LOG_FUNCTION_LEAVE</td>
<td>0x04</td>
<td>Logs the exits from the PCAN-Basic API functions.</td>
</tr>
<tr>
<td>Int32</td>
<td>LOG_FUNCTION_WRITE</td>
<td>0x08</td>
<td>Logs the CAN messages passed to the</td>
</tr>
</tbody>
</table>
**CAN_Write** function.

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>LOG_FUNCTION_READ</td>
<td>0x10</td>
<td>Logs the CAN messages received within the <strong>CAN_Read</strong> function.</td>
</tr>
<tr>
<td>Int32</td>
<td>LOG_FUNCTION_ALL</td>
<td>0xFFFF</td>
<td>Logs all possible information within the PCAN-Basic API functions.</td>
</tr>
</tbody>
</table>

*These values can be combined using the bitwise inclusive OR operator

**Trace-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>TRACE_FILE_SINGLE</td>
<td>0x00</td>
<td>Stores messages in a single file until the configured file size is reached.</td>
</tr>
<tr>
<td>Int32</td>
<td>TRACE_FILE_SEGMENTED</td>
<td>0x01</td>
<td>Stores messages distributed in several files.</td>
</tr>
<tr>
<td>Type</td>
<td>Constant</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Int32</td>
<td>TRACE_FILE_DATE</td>
<td>0x02</td>
<td>Includes date information into the name of the trace file.</td>
</tr>
<tr>
<td>Int32</td>
<td>TRACE_FILE_TIME</td>
<td>0x04</td>
<td>Includes the time information into the name of the trace file.</td>
</tr>
<tr>
<td>Int32</td>
<td>TRACE_FILE_OVERWRITE</td>
<td>0x08</td>
<td>Forces the overwriting of available traces (same name).</td>
</tr>
</tbody>
</table>

*These values can be combined using the bitwise inclusive OR operator*

**Feature values***:

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>FEATURE_FD_CAPABLE</td>
<td>1</td>
<td>The PCAN Channel represents a device that supports flexible data rate (CAN-FD).</td>
</tr>
<tr>
<td>Int32</td>
<td>FEATURE_DELAY_CAPABLE</td>
<td>2</td>
<td>The PCAN Channel represents a</td>
</tr>
</tbody>
</table>
device that supports the configuration of a delay between sending frames (FPGA devices only)

*These values can be combined using the bitwise inclusive OR operator

### Service-Status values:

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>SERIVCE_STATUS_STOPPED</td>
<td>1</td>
<td>The service is not running.</td>
</tr>
<tr>
<td>Int32</td>
<td>SERVICE_STATUS_RUNNING</td>
<td>4</td>
<td>The service is running.</td>
</tr>
</tbody>
</table>

#### Remarks

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

**Python:**

The definitions of these values have the following form: `TPCANParameter(value)` where `value` is the value contained in the column with the same name. e.g. `LOG_FUNCTION_ENTRY` is defined as `TPCANParameter(0x01)`.

#### See Also
FD Bit rate Parameter Definitions

Defines the different configuration parameters used to create a Flexible Data rate string for FD capable PCAN-Channels initialization. These values are used as parameter with CAN.InitializeFD (class-method: InitializeFD).

Clock Frequency parameters:

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>PCAN_BR_CLOCK</td>
<td>&quot;f_clock&quot;</td>
<td>Clock frequency Hertz (80000000, 60000000, 40000000, 30000000, 24000000, 20000000)</td>
</tr>
<tr>
<td>String</td>
<td>PCAN_BR_CLOCK_MHZ</td>
<td>&quot;f_clock_mhz&quot;</td>
<td>Clock frequency Megahertz (80, 60, 40, 30, 24, 20)</td>
</tr>
</tbody>
</table>

Nominal Bit rate parameters:

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>PCAN_BR_NOM_BRP</td>
<td>&quot;nom_brp&quot;</td>
<td>Clock prescaler for nominal quantum (1..1024).</td>
</tr>
</tbody>
</table>
### Data Bit rate parameters:

<table>
<thead>
<tr>
<th>Type</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>PCAN_BR_NOM_TSEG1</td>
<td>&quot;nom_tseg1&quot;</td>
<td>TSEG1 segment for nominal bit rate in time quanta (1..256).</td>
</tr>
<tr>
<td>String</td>
<td>PCAN_BR_NOM_TSEG2</td>
<td>&quot;nom_tseg2&quot;</td>
<td>TSEG2 segment for nominal bit rate in time quanta (1..128).</td>
</tr>
<tr>
<td>String</td>
<td>PCAN_BR_NOM_SJW</td>
<td>&quot;nom_sjw&quot;</td>
<td>Synchronization Jump Width for nominal bit rate in time quanta (1..128).</td>
</tr>
<tr>
<td>String</td>
<td>PCAN_BR_DATA_BRP</td>
<td>&quot;data_brp&quot;</td>
<td>Clock prescaler for fast data time quantum (1..1024).</td>
</tr>
<tr>
<td>String</td>
<td>PCAN_BR_DATA_TSEG1</td>
<td>&quot;data_tseg1&quot;</td>
<td>TSEG1 segment for fast data bit rate in time quanta (1..32).</td>
</tr>
<tr>
<td>String</td>
<td>PCAN_BR_DATA_TSEG2</td>
<td>&quot;data_tseg2&quot;</td>
<td>TSEG2 segment for fast data bit rate in time quanta (1..16).</td>
</tr>
<tr>
<td>String</td>
<td>PCAN_BR_DATA_SJW</td>
<td>&quot;data_sjw&quot;</td>
<td>Synchronization Jump Width for fast data bit rate in time quanta (1..16).</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------</td>
<td>------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
</tbody>
</table>

### Remarks

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

Following points are to be respected in order to construct a valid FD Bit rate string:

- The string must contain only one of the two possible "Clock Frequency" parameters, depending on the unit used (Hz, or MHz).
- The frequency to use must be one of the 6 listed within the "Clock Frequency" parameters.
- The value for each parameter must be separated with a '='.
  **Example:** "data_brp=1"
- Each pair of parameter/value must be separated with a ','. Blank spaces are allowed but are not necessary. Example:
  "f_clock_mhz=24, nom_brp=1,"
- Both Bit rates, or only the nominal one, must be defined within the string (PCAN_BR_DATA_* and PCAN_BR_NOM_*, or only PCAN_BR_NOM_*).

**Example with nominal Bit rate only:**

A valid string representing 1 Mbit/sec for both, nominal and data Bit rates:

"f_clock_mhz=20, nom_brp=5, nom_tseg1=2, nom_tseg2=1, nom_sjw=1"

**Example with nominal and data Bit rate:**

A valid string representing 1 Mbit/sec for nominal Bit rate, and 2 Mbit/sec for data Bit rate:
Parameter Value Ranges:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_clock</td>
<td>[80000000, 60000000, 40000000, 30000000, 24000000, 20000000]</td>
</tr>
<tr>
<td>f_clock_mhz</td>
<td>[80, 60, 40, 30, 24, 20]</td>
</tr>
<tr>
<td>nom_brp</td>
<td>1 .. 1024</td>
</tr>
<tr>
<td>nom_tseg1</td>
<td>1 .. 256</td>
</tr>
<tr>
<td>nom_tseg2</td>
<td>1 .. 128</td>
</tr>
<tr>
<td>nom_sjw</td>
<td>1 .. 128</td>
</tr>
<tr>
<td>data_brp</td>
<td>1 .. 1024</td>
</tr>
<tr>
<td>data_tseg1</td>
<td>1 .. 32</td>
</tr>
<tr>
<td>data_tseg2</td>
<td>1 .. 16</td>
</tr>
<tr>
<td>data_sjw</td>
<td>1 .. 16</td>
</tr>
</tbody>
</table>

Python:

The definitions of these values have the following form:

TPCANBitrateFD(string) where string is the value contained in the "value" column. e.g. PCAN_BR_CLOCK is defined as TPCANBitrateFD("f_clock").
See Also

CAN_InitializeFD (class-method: InitializeFD)
PCAN is the platform for PCAN-Basic. In the following topics there is an overview of PCAN and the fundamental practice with the interface DLL CanApi4 (PCAN-API)

In this Chapter

<table>
<thead>
<tr>
<th>Topics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PCAN Fundamentals</strong></td>
<td>This section contains an introduction to PCAN.</td>
</tr>
<tr>
<td><strong>PCAN-Light</strong></td>
<td>This section contains information about the previous version of the PCAN-Basic.</td>
</tr>
<tr>
<td><strong>PCAN-API</strong></td>
<td>This section compares the PCAN-Basic and CanApi4 interfaces, with a function description of CanApi4.</td>
</tr>
<tr>
<td><strong>Error Frames</strong></td>
<td>This section contains information about CAN Error frames.</td>
</tr>
<tr>
<td><strong>Log File Generation</strong></td>
<td>This section contains information about logging debug data within PCAN-Basic.</td>
</tr>
</tbody>
</table>
PCAN is a synonym for PEAK CAN APPLICATIONS and is a flexible system for planning, developing and using Controller Area Networks (CAN). It is a powerful product for both the developer and the end-user.

The PCAN system consists of a collection of Windows Device Drivers. These allow the Real-time connection of Windows applications to all CAN busses that are physically connected to the PC via a PCAN hardware. The interface to the user and the manager of a CAN-equipped installation are the so-called PCAN clients. With their help process factors are controlled and visualized. The drivers permit the connection of several clients, which are then able to communicate via the CAN busses. Furthermore, several hardware components are supported, which are based on the CAN controller Philips SJA1000.

So-called Nets are available. These specify virtual CAN busses that are extended into the PC. Several clients can connect to a virtual CAN bus. The connection to the outside world (a physical CAN bus) is possible with a hardware interface, e.g. the PCAN-Dongle or the PCAN-ISA card. The following figures give an overview of possible configurations.
Following rules apply to PCAN clients, nets and hardware:

- A PCAN client can be connected to more than one net.
- A net supplies several PCAN clients.
- A hardware component belongs to not more than one Net.
- A Net can have no hardware.
- If a Client sends a message, it will be transferred to every other Client and to the external CAN bus via the hardware.
• If a message is received over the hardware, it is received by every client. Every client receives only those messages, which pass its acceptance filter.
• Definition of the installed hardware and of the nets. Per hardware several nets may be defined. But only one net can be active.
• Clients connect to a net using the net name.
• Every PCAN client has a transmission queue, where CAN messages to be transmitted are waiting until the individual transmission time. At occurrence of the transmission time they are written into the transmission queue of the PCAN hardware.
• Every hardware contains a receive queue for buffering received CAN messages.

See Also

Understanding PCAN-Basic
PCAN-Light is the previous version of the [PCAN-Basic](#), the small API variant to [PCAN-API](#). It makes a fast and comprehensive working possible with the CAN bus system. The connection between the application program and the PCAN hardware is made by the appropriate device driver. The following illustration gives a short overview.
Following rules apply to PCAN-Light client (application software) and PCAN hardware:

- A client is in each case assigned to a hardware at each time. A
connection with one client to different hardware types is not possible at the same time.
- If a multi-channel hardware is present, each channel is treated like a separate hardware.
- If a Client sends a message, it will be transferred to every other Client and to the external CAN bus via the hardware.
- If a PCAN hardware is connected to a client, then no further client can access the same hardware.
- If a message is received over the hardware, it is received by every client. Every client receives only those messages, which pass its message (acceptance) filter.
- Every PCAN-Light client has a transmission queue, where CAN messages to be transmitted are waiting until the individual transmission time. At occurrence of the transmission time they are written into the transmission queue of the PCAN hardware.
- Every hardware contains a transmission queue for buffering CAN messages to be transmitted.

Schematic process flow of an PCAN-Light application software

<table>
<thead>
<tr>
<th>Start</th>
<th>Operate</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init-function to establish connection between non-Plug-and-Play hardware and application software.</td>
<td>Basic functions and additional functions</td>
<td>Call Close function</td>
</tr>
<tr>
<td>Init-function to establish connection between Plug-and-Play hardware and application software.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Start: Call the Init function to initiate the hardware.
2. Operate: After a successful start, the message filter could be adapted at your own conception. Furthermore CAN messages can be read and written. The driver of a type of hardware defined
the range of functions and is structured in basic and additional functions.

3. Finish: Call the Close function. This process disconnect the application software from PCAN hardware.

See Also

PCAN Fundamentals
Also called CanApi4 interface, is a synonym for CAN Application Programming Interface (version 2) and is a comprehensively programming interface to the PCAN system of the company PEAK-System Technik GmbH. This interface is more comprehensive than PCAN-Basic.

Important difference to PCAN-Basic:

- Transmit a CAN message at a fixed point of time.
- Several application programs could be connected to one PCAN-PC hardware.
- Detailed information to PCAN-PC hardware and the PCAN system (PCAN net and PCAN client).
- The PCAN client is connected via the net to the PCAN-PC hardware.

The following text is a short overview to the CanApi2 functions. The functions itself can be categorized as follows:

### Control

Register and remove functions for nets and hardware.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloseAll</td>
<td>Disconnects all hardware, nets, and clients.</td>
</tr>
<tr>
<td>RegisterHardware</td>
<td>Registers a Not-Plug-and-Play CAN hardware.</td>
</tr>
<tr>
<td>RegisterHardwarePCI</td>
<td>Registers a PCI CAN hardware.</td>
</tr>
<tr>
<td>RegisterNet</td>
<td>Defines of a PCAN net.</td>
</tr>
<tr>
<td>RemoveHardware</td>
<td>Removes and deactivates CAN hardware.</td>
</tr>
<tr>
<td>RemoveNet</td>
<td>Removes a PCAN net.</td>
</tr>
</tbody>
</table>

**Configuration**

Configuration functions for nets and hardware.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetDeviceName</td>
<td>Sets the PCAN device to be used for subsequent CanApi2 function calls.</td>
</tr>
<tr>
<td>SetDriverParam</td>
<td>Configures a driver parameter, eg. the size of the receive or transmit buffer.</td>
</tr>
<tr>
<td>SetHwParam</td>
<td>Configures a hardware parameter, eg. - the PEAK serial number, - and additional parameters for the PCAN-USB hardware.</td>
</tr>
<tr>
<td>SetNetParam</td>
<td>Configures net parameter.</td>
</tr>
</tbody>
</table>

**Client**

Functions for the management of the clients.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectToNet</td>
<td>Connects a client to a PCAN net.</td>
</tr>
<tr>
<td>DisconnectFromNet</td>
<td>Disconnects a client from a PCAN net.</td>
</tr>
<tr>
<td>RegisterClient</td>
<td>Registers an application as PCAN client.</td>
</tr>
<tr>
<td>RegisterMsg</td>
<td>Expands the reception filter of a client.</td>
</tr>
<tr>
<td>RemoveAllMsgs</td>
<td>Resets the filter of a Client for a connected Net.</td>
</tr>
<tr>
<td>RemoveClient</td>
<td>Removes a client from the driver.</td>
</tr>
<tr>
<td>ResetClient</td>
<td>Resets the receive and transmit queue of a</td>
</tr>
</tbody>
</table>
Functions for the data interchange over the CAN bus.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Reads a number of CAN_*-records from the client's receive queue. Records are CAN messages, error events, and other information.</td>
</tr>
<tr>
<td>Write</td>
<td>Writes a number of CAN messages or other commands into the transmit queue of a client.</td>
</tr>
</tbody>
</table>

Functions for the information about clients, nets, drivers, and hardware.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetClientParam</td>
<td>Retrieves client parameter, eg. - total number of transmitted or received CAN messages, - the PCAN driver name, PCAN net, or</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PCAN client name</td>
<td>- the number of received bits.</td>
</tr>
<tr>
<td>GetDeviceName</td>
<td>Retrieves the currently used PCAN device.</td>
</tr>
<tr>
<td>GetDiagnosticText</td>
<td>Reads the diagnostic text buffer.</td>
</tr>
<tr>
<td>GetDriverName</td>
<td>Retrieves the name of a PCAN device type.</td>
</tr>
<tr>
<td>GetDriverParam</td>
<td>Retrieves a driver parameter.</td>
</tr>
<tr>
<td>GetErrText</td>
<td>Translates an error code into a text.</td>
</tr>
<tr>
<td>GetHwParam</td>
<td>Retrieves a hardware parameter.</td>
</tr>
<tr>
<td>GetNetParam</td>
<td>Retrieves a net parameter.</td>
</tr>
<tr>
<td>GetSystemTime</td>
<td>Gets the system time.</td>
</tr>
<tr>
<td>Msg2Text</td>
<td>Creates a text form of a CAN message.</td>
</tr>
<tr>
<td>GetHardwareStatus</td>
<td>Detects the current status of a CAN hardware.</td>
</tr>
<tr>
<td>GetVersionInfo</td>
<td>Reads version and copyright information from the driver.</td>
</tr>
</tbody>
</table>
Error Frames can be received if:

- the hardware represented by the connected PCAN-Channel supports error frames generation,
- the connected PCAN-Channel configured the parameter PCAN_ALLOW_ERROR_FRAMES using the function CAN_SetValue (class-method: SetValue) to activate the error frames.

The error frame data will be placed into a TPCANMsg or TPCANMsgFD structure, according to the kind of initialization used, and its MSGTYPE field will have the PCAN_MESSAGE_ERRFRAME bit set.

The following list shows the elements of an error frame:

<table>
<thead>
<tr>
<th>Field</th>
<th>Information from...</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>ECC register&lt;br&gt;SJA1000 (Error Code Capture), bits 6 and 7 (ERRC0, ERRC1)</td>
<td>0: The message only transports updated Error Counter values&lt;br&gt;1: Bit error&lt;br&gt;2: Form error&lt;br&gt;4: Stuff error&lt;br&gt;8: Other type of error</td>
</tr>
<tr>
<td>DATA [0]</td>
<td>ECC register, bit 5 (DIR)</td>
<td>0: Error has occurred during transmission&lt;br&gt;1: Error has occurred during reception</td>
</tr>
<tr>
<td>DATA [1]</td>
<td>ECC register, bits 0 to 4 (SEG0 to SEG4)</td>
<td>Current position of the bit stream processor:&lt;br&gt;2: ID.28 to ID.21&lt;br&gt;3: Start of frame&lt;br&gt;4: Bit SRTR&lt;br&gt;5: Bit IDE&lt;br&gt;6: ID.20 to ID.18</td>
</tr>
</tbody>
</table>
### Remarks

For further information about this, see Philips Data Sheet "SJA1000 Stand-alone CAN controller".

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA [3]</td>
<td>TX Error Counter Register (TXERR)</td>
<td>Current value of the Transmit Error counter</td>
</tr>
</tbody>
</table>
Log File Generation

In order to support debugging of problems, that can arise during CAN communication, PCAN-Basic can generate a log file, containing a protocol of all API function calls. There are two different ways to configure and activate this logging functionality:

- Using the API.
- Using the Windows Registry.

## Log configuration using API

In order to configure the Log functionality, PCAN-Basic API provides 3 parameters that can be configured with the function `CAN_SetValue` (class-method: `SetValue`). These parameters are:

- `PCAN_LOG_LOCATION`, to set the location of the log file.
- `PCAN_LOG_CONFIGURE`, to configure the content of the log file.
- `PCAN_LOG_STATUS`, to enable/disable logging.

### Example:

Within the following example, the PCAN-Basic is configured to log all possible information (LOG_FUNCTION_ALL = 0xFFFF), as well as to store the log file on the desktop of an user called "admin".

**C++:**

```cpp
TPCANStatus result;
char strMsg[MAX_PATH];

// Configures the data in the log file.
//
iBuffer = LOG_FUNCTION_ALL;
result = CAN_SetValue(PCAN_NONEBUS, PCAN_LOG_CONFIGURE);
if(result != PCAN_ERROR_OK)
```
// Configures the path of the log file.
//
char *buffer = "C:\\users\\admin\\desktop";
result = CAN_SetValue(PCAN_NONEBUS, PCAN_LOG_LOCATION)
if(result != PCAN_ERROR_OK)
{
    CAN_GetErrorText(result, 0, strMsg);
    MessageBox(strMsg);
}

// Configures the status of the log file
//
int iBuffer = PCAN_PARAMETER_ON;
result = CAN_SetValue(PCAN_NONEBUS, PCAN_LOG_STATUS)
if(result != PCAN_ERROR_OK)
{
    CAN_GetErrorText(result, 0, strMsg);
    MessageBox(strMsg);
}

\* Log configuration using Windows Registry

In order to enable the log file generation, the following registry key must be created:

HKEY_CURRENT_USER\SOFTWARE\PEAK-System\PCAN-Basic\%

The existence of this key is analogous to use the function CAN_SetValue (class-method: SetValue) to set the parameter PCAN_LOG_STATUS to "on". If this key is not present, then no log file is generated.
Configuration:

If no further configuration is made, then the default values for PCAN_LOG_LOCATION, and PCAN_LOG_CONFIGURE are used. In order to configure the location and content of the log file, two registry values are used:

Flags: This is a DWORD value, that represents a logical OR operation between the values LOG_FUNCTION_* that are wanted to be included within the logging data. The value LOG_FUNCTION_ALL causes logging all possible information.

Path: This is a String value, that represents the path to a folder in the computer, where the log file will be created.

Example:

Within the following example, the PCAN-Basic is configured to log function entries (LOG_FUNCTION_ENTRY = 1), function parameters (LOG_FUNCTION_PARAMETERS = 2), and function outs (LOG_FUNCTION_LEAVE = 4), as well as to store the log file on the desktop of an user called "admin".

```
[HKEY_CURRENT_USER\SOFTWARE\PEAK-System\PCAN-Basic]
"Flags"=dword:00000007
"Path"="C:\Users\admin\desktop"
```

Remarks:

The registry key should be deleted (or renamed) after a debug session is done. If the key is lefted, all PCAN-Basic applications running under the same user account will remain writing data to their log files, generating in this way huge text files that consume hard-disk space unnecessarily.
Welcome to the documentation of PCAN-Basic, the new small Version of the PCAN-API from PEAK-System.

- **Introduction**  »  **DLL API Reference**  »  **Additional Information**

_Last Update: 31.07.2017_