Software Documentation

Windows CE CAN Interface
Software Interface for .NET

Version 1.03
2003-12-07

Elektronik Systeme

Untere Waldplätze 23
D-70569 Stuttgart
Fon: +49(0)711-123722-0
Fax: +49(0)711 – 123722-99

History

Date	V	Platform	A,M,R	Chapter	Description	Au
2014-12-08	1.03		M	*	Changed to new Company CI	JG

V	Version
A,M,R	Added, Modified, Removed
Au	Author

About this document

This document describes how to handle the CAN Interface under Windows CE.

Table of Contents

1 Introduction

Some NetDCU boards offer on-board CAN ports for direct integration of the NetDCU into a CAN network. Other NetDCUs can use the extension board NDCU-ADP/CAN2 that provides two CAN ports.
The `.NET` interface to this CANINTF driver is done by a C# class `CanPort`, wrapping around the driver interface and making the native Win32 functions available to the `.NET` languages like `C#.NET` and `VisualBasic.NET`. The class allows the usage of managed data types and does all conversions required to call the driver without the user having to bother about the details.
This document describes all functions and data types provided by the `CanPort` class. In addition we introduce a special exception class `CanPortException`, allowing easy error handling in combination with the `CanPort` class. At the end we have included two sample programs `CanWrite` and `CanRead`, showing the usage of the `CanPort` class.

Remark
In the remaining document we'll use the term "NetDCU" as generic reference to all our Windows CE boards. This should also include PicoMOD boards where appropriate.

2 Installation

2.1 Installing the device driver

For using the CAN ports, it is required to install the CANINTF driver. This is documented in “WINCE-CAN-Interface, Software Documentation, NetDCU-ADP/CAN2”. This is also required when using the CAN port with the `.NET` environment.

2.2 Installing the .NET library CanPort.dll

To use the `CanPort.dll` library for `.NET`, you have to copy it to your PC, for example to your Visual Studio project directory, and add a reference to it in your project. This can be done in two ways:

1. In the solution explorer, right click on the “References” entry and select “Add Reference...”
2. In menu “Project” select “Add Reference...”

In both cases you will be presented with a dialog having several tabs. Click on the tab “Browse” and search for the `CanPort.dll` in your project directory. After clicking OK, entry “CanPort” will appear in the References section of the Solution Explorer.
If the `CanPort` class is not automatically recognized in the editor immediately, close and re-open your solution. Now the new objects should be supported by the editor.

3 The CanPort class

The `CanPort` class defines all functions needed for accessing the CAN port, including some data types, constants and enumerations. The class is embedded in the `FS.NetDCU` name-space, so the fully qualified name is `FS.NetDCU.CanPort`.
First we will describe the member functions, then follow the data types used together with them in the second part of this chapter.

Error Handling
As with most low-level Windows drivers written in C, it is common for a function to return an error or success value as the direct return value and return any requested data in data structures passed by reference as parameters. Contrary to this, modern languages like C# usually use asynchronous exceptions to report failure and therefore can use the return value directly to transfer the requested data, usually as objects.
With the `CanPort` class, we let you choose what behaviour you want. By default any error in a `CanPort` function will throw a `CanPortException`. However you can change this behaviour by calling `HandleErrorsViaReturn()` immediately after constructing the `CanPort` object. This switches this instance to the C style convention and then each function returns 0 for success and an error value different from 0 for failure.
The two sample programs at the end of this document show the usage of the two different methods. `CanWrite` uses the exception technique, `CanRead` the return value method.

3.1 CanPort() (Construction)

Signature:
`CanPort(string FileName, CanAccess access)`

Parameters:

`FileName`	Name of the device (`CID1:`, `CID2:`)
`access`	Access type: Device query access, read access, write access, or read-write access.

Description:
Open the device file. Throw a `CanPortException` if it fails. The device file is automatically closed by the destructor when the object is destroyed.
The CAN ports have the device names `CID<n>:` where `<n>` is the number of the port, usually 1 or 2. The access defines whether you want to transmit or receive messages.
For the description of `CanAccess` see page 28.

Example:

```
try
{
     //                  Create                  a                  CanPort                  object
     CanPort          pCAN          =          new          CanPort("CID1:",          CanPort.CanAccess.READ);
}
catch                                        (CanPortException                                        e)
{
     //              Handle              error              according              to              e.Reason
}
```

3.2 Read()

Signature:
```
int Read(out string result)
```

Parameters:

result	String that was read

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Read an event entry from the device file. If no event is currently in the queue, wait until an event arrives or until a timeout occurs. You can set the timeout values with `SetCommTimeouts()`.
When using this function, the binary data of the CAN port events and messages is converted to a textual representation that can easily be written to log files or printed to the display. In fact there are two conversions involved: first a binary to ASCII text conversion within the CAN driver, and then an ASCII to Unicode conversion within the `Read()` wrapper function.
It is possible to parse and analyse these text contents to detect the type of event and react accordingly, but this can more easily achieved by using `ReadEventData()` instead. This also avoids the overhead of the text conversions.

Depending on the event that occurred on the CAN bus, the output of `Read()` can have one of the following appearances:
Event RECEIVED:

received\t<time_high>:<time_low>\t<id>\t<rtr>\t<dlc>\r\n
\t<msg0>\t<msg1>...\t<lost>\n

Event TRANSMITTED:

transmitted\t<time_high>:<time_low>\t<id>\t<rtr>\t<dlc>\r\n
\t<msg0>\t<msg1>...\n

Event BUS_ERROR:

bus error\t<time_high>:<time_low>\n

Event WARNING:

warning\t<time_high>:<time_low>\n

Event ARBITRATION_LOST:

arbitration lost\t<time_high>:<time_low>\n

Event OVERRUN:

overrun\t<time_high>:<time_low>\n

Event LEAVING_STANDBY:

leaving standby\t<time_high>:<time_low>\n

Event ENTERING_STANDBY:

entering standby\t<time_high>:<time_low>\n

Event PASSIVE:

passive\t<time_high>:<time_low>\n

Event DEVICE_CHANGED:

device changed\t<time_high>:<time_low>\n

Any other event:

unknown event\t<time_high>:<time_low>\n

Here the parameters have the following meaning:

<time_high>	Highword of the time when the event occurred (32 bit as decimal value)
<time_low>	Lowword of the time when the event occurred (32 bit as decimal value)
<id>	ID of the message, usually identifying the target device or message type (32 bit value as 8 hex digits)
<rtr>	Remote transmission request (decimal value) 0: CAN message with data 1: Request data from receiver
<dlc>	Data length code (0..8 as decimal value)
<msg0>	First message byte (8 Bit as 2 hex digits)
<msg1> ...	Remaining message bytes (each 2 hex digits); there are exactly <dlc> bytes in total
<lost>	Number of lost messages since last Read() (32 bit as decimal value)
\t	Tabulator character <TAB> (=0x09)
\r	Carriage return character <CR> (=0x0D)
\n	Line feed character <LF> (=0x0A)

Lost messages may occur if the messages arrive faster than they are read and the internal buffer does not suffice.

Example:

```
received          0:4018194          000003d7          0          6
 18 29 3A 3B 4C 5D 0
```

This shows that a message was received at CAN bus time 0:4018194 with ID 0x000003d7, with remote transmission request 0, and data length code 8, resulting in the eight message bytes 0x18, 0x29, 0x3A, 0x3B, 0x4C and 0x5D. There were no messages lost.

3.3 Write()

Signature 1:
`int Write(string tosend)`

Parameters:

`tosend`	Message to send

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Write a message to the device file with the following syntax:
`<id> <rtr> <dlc> <msg0> <msg1> ... \n`
Every entry is a hex number:

`<id>`	ID of the message, usually identifying the target device or message type
`<rtr>`	Remote transmission request 0: CAN message with data 1: Request data from receiver
`<dlc>`	Data length code, i.e. number of message values following (0..8)
`<msg0>`	The first message byte value (0..FF)
`<msg1>`	The second message byte value (0..FF). Up to eight message bytes can be given.
`\n`	Line feed character `<LF>` (=0x0A)

Please note that `Write()` is text based. This is convenient in many ways, but this also means the CAN bus driver must interpret the text and convert the values back to binary data. Moreover, the `.NET` environment uses Unicode characters, but the driver expects ASCII characters. So the `Write()` wrapper function must convert Unicode to ASCII first before calling the CAN driver.
You can avoid this overhead by using `WriteTransmitData()` instead, which provides a binary interface to the CAN interface driver.

Example:
Send three bytes 0x13, 0xB4, 0xCF with message ID 0x1A7 and remote transmission request 0.

```
CanPort pCAN = new CanPort(...);
pCAN.Write("0x1A7 0 3 13 B4 CF\n");
```

Signature 2:
`int Write(uint id, byte dlc, byte[] msg)`

Parameters:

`id`	Message ID
`dlc`	Data length code (0..8)
`msg`	Array containing the message bytes; it must have at least `dlc` entries!

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Convert the parameters to a message string and send it to the CAN port device. `rtr` is implicitly taken as 0. Here the byte array can be larger than the message, but only the first `dlc` bytes are actually transmitted.
Please note that `Write()` always uses the text interface to talk to the CAN driver (see page 10). If you want a binary interface, consider using `WriteTransmitData()` instead.

Example:
Sending 3 bytes 0x13, 0xB4, 0xCF with message ID 0x1A7.

```
CanPort pCAN = new CanPort(...);
byte[] msg = new byte[6] {0x13, 0xB4, 0xCF, 0, 0, 0};
pCAN.Write(0x1A7, 3, msg);
```

Signature 3:
```
int Write(uint id, byte[] msg)
```

Parameters:

`id`	Message ID
`msg`	Array containing the message bytes; it must be 0 to 8 bytes of length

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Convert the parameters to a message string and send it to the CAN port device. The message length is taken from the array length, `rtr` is implicitly taken as 0.
Please note that `Write()` always uses the text interface to talk to the CAN driver (see page 10). If you want a binary interface, consider using `WriteTransmitData()` instead.

Example:
Send three bytes 0x13, 0xB4, 0xCF with message ID 0x1A7.
```
CanPort pCAN = new CanPort(...);
byte[] msg = new byte[] {0x13, 0xB4, 0xCF};
pCAN.Write(0x1A7, msg);
```

Signature 4:
```
int Write(uint id)
```

Parameters:

id	Message ID

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Send a message that requests an answer from some target device. Please note that this signature implies an empty message (`dlc=0`) and sets the remote transmission request (`rtr=1`).
Please note that `Write()` always uses the text interface to talk to the CAN driver (see page 10). If you want a binary interface, consider using `WriteTransmitData()` instead.

Example:
Send a message with ID code 0x1B9 that requests some information from a target device handling this ID code.
```
CanPort pCAN = new CanPort(...);
pCAN.Write(0x1B9);
```

3.4 SetCommTimeouts()

Signature:
```
int SetCommTimeouts(ref CommTimeouts timeouts)
```

Parameters:

timeouts	Timeout values to be used on CAN bus

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Set the timeout values for this CAN bus device. This is similar to setting timeout values for a serial line.
For a description of `CommTimeouts` see page 31.

3.5 GetCommTimeouts()

Signature:
```
int GetCommTimeouts(out CommTimeouts timeouts)
```

Parameters:

`timeouts`	Current timeout values

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Get the current timeout values of this CAN bus device.
For a description of `CommTimeouts` see page 31.

3.6 SetCommMask()

Signature:
`int SetCommMask(CanEventFlags mask)`

Parameters:

`mask`	Mask of reported events

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Configure which CAN bus events are reported and which are ignored.
For a description of `CanEventFlags` see page 29.

Example:
```
CanPort pCAN = new CanPort(...);
pCAN.SetCommMask(CanPort.CanEventFlags.CANBUS_TRANSFERS
                 |                                          CanPort.CanEventFlags.BUS_ERROR
                 | CanPort.CanEventFlags.WARNING);
```

3.7 GetCommMask()

Signature:
`int GetCommMask(out CanEventFlags mask)`

Parameters:

`mask`	Current mask of reported events

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Get the currently allowed events of this CAN bus device. This is a combination of one or more `CanEventFlags`.
For a description of `CanEventFlags` see page 29.

3.8 WaitCommEvent()

Signature:
```
int WaitCommEvent(out CanEventFlags mask)
```

Parameters:

`mask`	Event that happened

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Wait until an event occurs. Which events are reported is set with `SetCommMask()`. The event that occurred is reported in `mask`.
For a description of `CanEventFlags` see page 29.

3.9 WriteAcceptanceFilter()

Signature:
```
int                                          WriteAcceptanceFilter(
          ref CanAcceptanceFilter filter)
```

Parameters:

`filter`	Combination of the allowed IDs; all other IDs are ignored

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Set the acceptance filter. The acceptance filter determines which incoming messages are accepted and which are ignored, depending on the message ID.
For a description of `CanAcceptanceFilter` and an example see page 32.

3.10 ReadAcceptanceFilter()

Signature:

```
int                                                 ReadAcceptanceFilter(
          out CanAcceptanceFilter filter)
```

Parameters:

`filter`	Currently active settings

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:

Returns the currently active acceptance filter. The acceptance filter determines which incoming messages are accepted and which are ignored, depending on the message ID.
For a description of `CanAcceptanceFilter` and an example see page 32.

3.11 SetBaudrate()

Signature:

```
int SetBaudrate(UInt32 baudrate)
```

Parameters:

`baudrate`	Transfer speed

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:

Set the speed of the CAN bus communication. The minimum and maximum allowed speed can be obtained by function `ReadProperties()`. Usually they are in the range of 20000 to 1000000 Hz.
If you want to set a speed from a predefined list of baud rates, use `SetBaudrateByIndex()` instead.

3.12 GetBaudrate()

Signature:
```
int GetBaudrate(out UInt32 baudrate)
```

Parameters:

`baudrate`	Current transfer speed

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Returns the current speed of the CAN bus communication.
There is another function `GetBaudrateByIndex()` that returns the index into a table of predefined baud rates.

3.13 SetBaudrateByIndex()

Signature:
```
int SetBaudrateByIndex(UInt32 index)
```

Parameters:

`index`	Index into predefined baud rate table

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Set the speed of the CAN bus communication by using the entry of the predefined baud rate table given by index. The baud rate table (and maximum index) can be obtained with `ReadProperties()`.
If you don't like to use a predefined speed, you can set the baud rate directly by using function `SetBaudrate()`.

3.14 GetBaudrateByIndex()

Signature:
```
int GetBaudrateByIndex(out UInt32 index)
```

Parameters:

`index`	Index into predefined baud rate table

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Returns the index of the entry of the predefined baud rate table that is corresponding with the current speed of the CAN bus communication. The baud rate table (and maximum index) can be determined with `ReadProperties()`.
If you want to get directly the speed, not the list index, use `GetBaudrate()` instead.

3.15 Init()

Signature:
```
int Init()
```

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Initialise the CAN bus.

3.16 SetCanMode()

Signature:
`int SetCanMode(CanMode mode)`

Parameters:

`mode`	Mode to set

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Set the working mode of the CAN controller.
Depending on the capabilities of the CAN bus controller, it can run in two different modes: BasiCan mode (also known as CAN2.0A) and PeliCan mode (also known as CAN2.0B). PeliCan mode is more powerful and allows additional features like an extended frame format with 29-bit identifiers. BasiCan mode can tolerate 29-bit identifiers on the bus, but can only process the normal frame format with 11-bit identifiers.
You can use `ReadProperties()` to determine what modes the NetDCU controller supports. For a description of `CanMode` see page 30.

3.17 SetCommand()

Signature:
`int SetCommand(CanCommand command)`

Parameters:

`command`	Command to execute

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Execute a special command on the CAN bus controller. You can use `ReadProperties()` to determine which are supported by the NetDCU controller.
For a description of `CanCommand` see page 29.

3.18 WriteTransmitData()

Signature 1:

```
int WriteTransmitData(CanTransmitFormat fmt,
    UInt32 id, byte rtr, byte dlc, byte[] msg)
```

Parameters:

`fmt`	Frame format (default, 11-bit, 29-bit IDs)
`id`	Message ID
`rtr`	Remote transmission request 0: CAN message with data 1: Request data from receiver
`dlc`	Data length code (0..8)
`msg`	Message (at least `dlc` bytes)

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:

Transmit the given message in the given format on the CAN bus. The `msg` array may contain more bytes, but only the first `dlc` bytes are used.

Unlike the `Write()` functions that use an intermediate text representation to talk to the driver, this function uses the binary data directly.

For a description of `CanTransmitFormat` see page 28.

Example:

Send three bytes 0x13, 0xB4, 0xCF with message ID 0x1A7 and remote transmission request 0 in the default frame format.

```
CanPort pCAN = new CanPort(...);
byte[] msg = new byte[6] {0x13, 0xB4, 0xCF, 0, 0, 0};
pCAN.WriteTransmitData(CanPort.CanTransmitFormat.DEFAULT,
                       0x1A7, 0, 3, msg);
```

Signature 2:

```
int WriteTransmitData(CanTransmitFormat fmt,
UInt32 id, byte rtr, byte[] msg)
```

Parameters:

`fmt`	Frame format (default, 11-bit, 29-bit IDs)
`id`	Message ID
`rtr`	Remote transmission request 0: CAN message with data 1: Request data from receiver
`msg`	Message (`dlc` determined from array length)

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:

Transmit the given message in the given format on the CAN bus. The message length is determined by the `msg` array length, so it must not contain more than 8 bytes.
Unlike the `Write()` functions that use an intermediate text representation to talk to the driver, this function uses the binary data directly.
For a description of `CanTransmitFormat` see page 28.

Example:

Send three bytes 0x13, 0xB4, 0xCF with message ID 0x1A7 in the default frame format.

```
CanPort pCAN = new CanPort(...);
byte[] msg = new byte[] {0x13, 0xB4, 0xCF};
pCAN.WriteTransmitData(CanPort.CanTransmitFormat.DEFAULT,
0x1A7, 0, msg);
```

Signature 3:

```
int WriteTransmitData(CanTransmitFormat fmt,
    ref CanTransmitData data)
```

Parameters:

`fmt`	Frame format (default, 11-bit, 29-bit IDs)
`data`	Message data to send

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:

Transmit the given message in the given format on the CAN bus. This variant uses a structure to hold the transmit data.

Unlike the `Write()` functions that use an intermediate text representation to talk to the driver, this function uses the binary data directly.

For a description of `CanTransmitFormat` see page 28, for a description of `CanTransmitData` see page 35.

Example:

Re-send some received message with the new ID 0x123.

```
CanPort pCAN = new CanPort(...);
CanPort.CanEvent e;
pCAN.ReadEventData(out e);
e.data.id = 0x123;
pCAN.WriteTransmitData(CanPort.CanTransmitFormat.DEFAULT,
    ref e.data);
```

3.19 ReadEventData()

Signature:
```
int ReadEventData(out CanEvent evnt)
```

Parameters:

`evnt`	Returned event data

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Read the data of the next event in the queue. This command generates an error when all events are already read and there is no new event data available. Usually this is combined in some form with `WaitCommEvent()`.
If you want to get the event data directly in some readable text form, you may want to use function `Read()` instead.
For a description of `CanEvent` see page 35.

Example:
```
CanPort pCAN = new CanPort(...);
CanPort.CanEvent e;
pCAN.ReadEventData(out e);
```

3.20 ReadTime()

Signature:
```
int ReadTime(out CanTime time)
```

Parameters:

`time`	Current CAN bus time

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Read the current CAN bus time. This is based on the internal tick count, i.e. the number of milliseconds since power-on. It is a 64 bit number divided in a low and high part.
For a description of `CanTime` see page 35.

3.21 SetDefaultFrameFormat()

```
int                                                SetDefaultFrameFormat(
                          CanTransmitFormat fmt)
```

Parameters:

`fmt`	Frame format (default, 11-bit, 29-bit IDs)

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Set the default frame format. The default format set here is used in all subsequent calls of `WriteTransmitData()`, when `DEFAULT` is used there.
For a description of `CanTransmitFormat` see page 28.
If `DEFAULT` is used here with `SetDefaultFrameFormat()`, the chosen frame length depends on the capabilities of the CAN bus controller:

CAN2.0A	Set default to `STANDARD` (11-bit IDs)
CAN2.0B	Set default to `EXTENDED` (29-bit IDs)

Call `ReadProperties()` to check the features of the CAN bus controller.

3.22 TestDevice()

Signature:

```
int TestDevice()
```

Return:

0	Normal Mode
1	Reset Mode
-1	Device not active

Description:
Test the mode of the CAN bus controller.
Please note that this function returns the result directly, there is no special error or success report.

3.23 ReadProperties()

Signature:
```
int ReadProperties(out CanProperties prop)
```

Parameters:

`prop`	Current properties

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Read the features of the CAN bus controller. This returns the device name, the driver software version, the capabilities of the CAN bus controller, the supported commands, the supported baud rates, and the number of controller registers.
For a description of `CanProperties` see page 36.

3.24 ReadRegister()

Signature:
```
int ReadRegister(byte reg, out byte val)
```

Parameters:

`reg`	Number of register to read from
`val`	Read value

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Read the specified CAN controller register in Normal Mode, i.e. switch to Normal Mode before reading.
The CAN controller can run in two different modes: Reset Mode automatically active after power-on, and Normal Mode for regular operation. In Reset Mode, some configuration registers may be accessible that are hidden in Normal Mode. You can use `TestDevice()` to determine the current running mode.

3.25 WriteRegister()

Signature:
```
int WriteRegister(byte reg, byte val)
```

Parameters:

`reg`	Number of register to write to
`val`	Value to write

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Write the given value to the specified CAN controller register in Normal Mode, i.e. switch to Normal Mode before writing.
The CAN controller can run in two different modes: Reset Mode automatically active after power-on, and Normal Mode for regular operation. In Reset Mode, some configuration registers may be accessible that are hidden in Normal Mode. You can use `TestDevice()` to determine the current running mode.

3.26 ReadRegisterRM()

Signature:
```
int ReadRegisterRM(byte reg, out byte val)
```

Parameters:

`reg`	Number of register to read from
`val`	Read value

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Read the specified CAN controller register in Reset Mode, i.e. switch to Reset Mode before reading.
The CAN controller can run in two different modes: Reset Mode automatically active after power-on, and Normal Mode for regular operation. In Reset Mode, some configuration registers may be accessible that are hidden in Normal Mode. You can use `TestDevice()` to determine the current running mode.

3.27 WriteRegisterRM()

Signature:
`int WriteRegisterRM(byte reg, byte val)`

Parameters:

`reg`	Number of register to write to
`val`	Value to write

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Write the given value to the specified CAN controller register in Reset Mode, i.e. switch to Reset Mode before writing.
The CAN controller can run in two different modes: Reset Mode automatically active after power-on, and Normal Mode for regular operation. In Reset Mode, some configuration registers may be accessible that are hidden in Normal Mode. You can use `TestDevice()` to determine the current running mode.

3.28 enum CanAccess

Values:

`QUERY`	Device query access only. You can't transmit or receive, just query the device settings.
`READ`	Read access. You can receive messages.
`WRITE`	Write access. You can transmit messages.
`READ_WRITE`	Read and write access. You can receive and transmit messages.

Description:
Defines the access type to the CAN port when constructing an instance of the `CanPort` class.

3.29 enum CanTransmitFormat

Values:

`DEFAULT`	Send in default frame format
`STANDARD`	Send frames with 11-bit IDs
`EXTENDED`	Send frames with 29-bit IDs

Description:
CAN bus transmission formats for `WriteTransmitData()` and `SetDefaultFrameFormat()`.

3.30 enum CanEventFlags

Values:

RECEIVED	Message received
TRANSMITTED	Message transmitted
CANBUS_TRANSFERS	Both of the above events
BUS_ERROR	There was a CAN bus error
WARNING	There was a warning
ARBITRATION_LOST	CAN bus arbitration lost
OVERRUN	Message overrun
CANBUS_ERRORS	All of the above four events
PASSIVE	Passive mode
ENTERING_STANDBY	Entering standby mode
LEAVING_STANDBY	Leaving standby mode
DEVICE_CHANGED	Device mode changed
CANBUS_STATES	All of the above four events
CANBUS_ALL	All possible events

Description:
These flags describe the possible events that can happen on the CAN bus. When setting a mask with `SetCommMask()`, any combination of the above values is possible. When waiting for an event with `WaitCommEvent()` or when reading event data with `ReadEventData()`, only one single event is active and therefore set.

3.31 enum CanCommand

Values:

ABORT_TRANSMISSION	Clear out queue
CLEAR_OVERRUN	Clear overrun flag
ENTER_STANDBY	Enter standby mode, wake up on any event
LEAVE_STANDBY	Manually leave standby mode
SELF_RECEPTION_REQUEST	Self reception request
LISTEN_ON	Enable listen-only mode
LISTEN_OFF	Disable listen-only mode
VIRTUALIZE_ON	Enter virtualization mode
VIRTUALIZE_OFF	Leave virtualization mode

Description:
These values can be used in `SetCommand()`. Use function `ReadProperties()` to determine which commands are supported by the CAN bus controller.

3.32 enum CanMode

Values:

BASICAN	Set BasiCan mode (=CAN2.0A)
CAN_2_0_A	The same as BASICAN
PELICAN	Set PeliCan mode (=CAN2.0B)
CAN_2_0_B	The same as PELICAN

Description:
These values are used in SetCanMode() and describe one of the possible CAN modes: BasiCan mode (=CAN2.0A) or PeliCan mode (=CAN2.0B).

3.33 enum CanChipsetFlags

Values:

CAN_2_0_A	Controller supports BasiCan mode (CAN2.0A)
CAN_2_0_B	Controller supports PeliCan mode (CAN2.0B)
EXT_FRAME	Controller supports extended frames
POLLING	Controller supports polling

Description:
This value is used in the CanProperties structure and describes the capabilities of the CAN controller. The values are flags, so the reported value can be any combination of the above values.

3.34 enum APIError

Values:

ERROR_FILE_NOT_FOUND	Port not found
ERROR_ACCESS_DENIED	Access to port denied
ERROR_INVALID_HANDLE	Invalid handle
ERROR_NOT_READY	Device not ready
ERROR_WRITE_FAULT	Write fault
ERROR_INVALID_PARAMETER	Bad parameters
ERROR_INVALID_NAME	Invalid port name

Description:
This type enumerates some of the error values you might encounter when using the CanPort class. The Win32 API functions usually return error values that can be checked by Marshal.GetLastWin32Error() and this is also the error code returned by almost all functions of the CanPort class on failure.
Please note that this list is not exhaustive and may be extended in future software versions.

3.35 struct CommTimeouts:

Entries:

`UInt32` `ReadIntervalTimeout`

Maximum acceptable time between two bytes on the CAN bus line. 0 means no interval timing used.

`UInt32` `ReadTotalTimeoutMultiplier`

Total read timeout multiplier. This number is multiplied with the requested number of bytes to read.

`UInt32` `ReadTotalTimeoutConstant`

This value is added to the product above to build the total read timeout.

`UInt32` `WriteTotalTimeoutMultiplier`

Total write timeout multiplier. This number is multiplied with the requested number of bytes to write.

`UInt32` `WriteTotalTimeoutConstant`

This value is added to the product above to build the total write timeout.

Description:

Timeout values for CAN access. All timeout values are given in milliseconds (ms).

3.36 struct CanAcceptanceFilter

Entries:

`UInt32 code`	ID code
`UInt32 mask`	Mask for active/inactive bits 1-bit: always accepted 0-bit: accepted if code matches

Description:

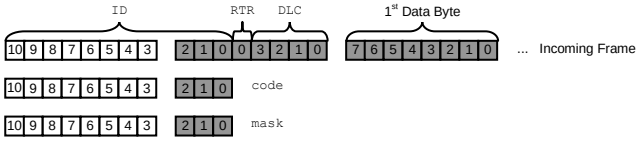
The acceptance filter defines which incoming messages are accepted and which are ignored. This depends on the message ID. By using a code part and a mask part, you can define ranges of IDs that are accepted by the CAN bus device.

The mask defines which bits of the message ID don't matter and are always accepted, and which bits are relevant and must match the code to be accepted. The check uses a binary `OR` of the mask and the code on one hand, and the mask and the ID on the other hand and then checks if the two values are equal. Then the message is accepted.

The handling of the mask differs depending on the CAN controller mode.

1. BasiCan mode (CAN2.0A)

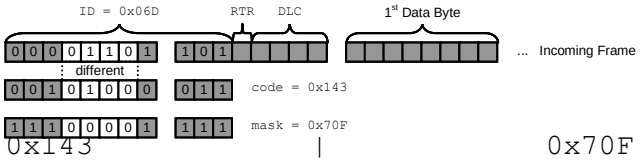
Here `code`, `mask` and message `ID` are 11 bits wide, but only the upper 8 bits of the mask can be influenced. The lower 3 bits are always assumed as 1. Therefore the lower 3 bits of the `ID` are always accepted, no matter what the `code` says. In the following graphic you can see how the `mask` and `code` match to the incoming frame format.



Grey squares denote fields having no influence on the ID acceptance.

Example 1:

`code = 0x143, mask = 0x70F, ID = 0x06F.`



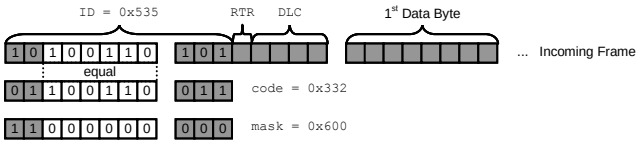
`0x143 | 0x70F = 0x74F`

`0x06D | 0x70F = 0x76F`

As the two values are different, the message is ignored.

Example 2:

`code = 0x332, mask = 0x600, ID = 0x535.`



First of all the lower 3 bits of the mask will automatically be set to 1, so in the end the `mask = 0x607` is used.

`0x332 | 0x607 = 0x737`

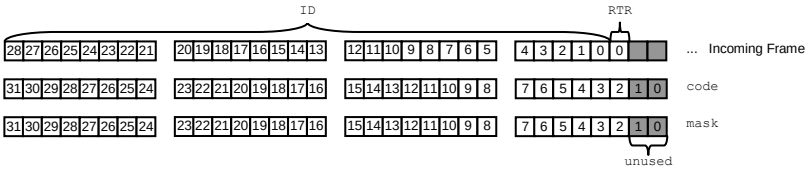
`0x535 | 0x607 = 0x737`

The two values are equal, so this message is accepted.

Remark: We used `mask = 0x600` just to show the behaviour of the three least significant bits. But please do not rely on this mechanism. It is recommended to always set these bits in the mask, so there is no surprise on future hardware. Therefore you should set `mask = 0x607` right from the beginning.

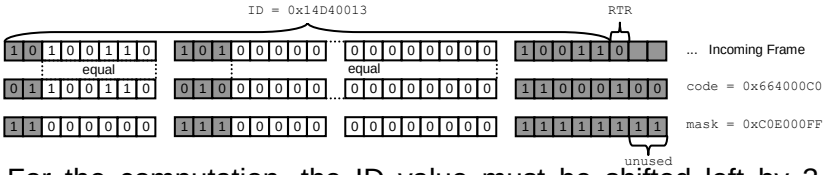
2. PeliCan mode (CAN2.0B), receiving extended frame

Here `code`, `mask` and message `ID` encompass 29 bits, but also RTR is used as a 30th bit in the acceptance computation. However these bits are set in the upper 30 bits of the 32 bit value, the lower two bits are unused and should be set to 1.



Example:

`code = 0x664000C0, mask = 0xC0E000FF, ID = 0xA6A000F, RTR = 0`



For the computation, the ID value must be shifted left by 3 and the RTR code must be inserted at bit 2. This results in `IDshifted = 0xA6A00098`.

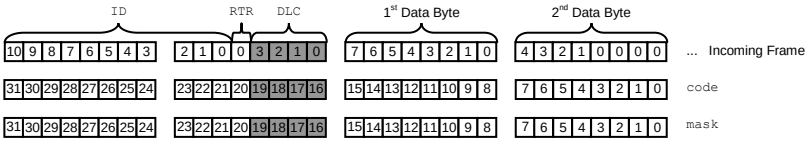
0x664000C0 | 0xC0E000FF = 0xE6E000FF

0xA6A00098 | 0xC0E000FF = 0xE6E000FF

The values are equal, the message is accepted.

3. PeliCan mode (CAN2.0B), receiving standard frame

Here the most significant 12 bits are matched against the message `ID` and `RTR`, and the least significant 16 bits are matched against the first two data bytes of the incoming message. Bits 16 to 19 of the mask are automatically taken as 1s, so these bits, mapping the `DLC` of the message, are always accepted.



Remark

The SJA1000 CAN bus controller chip, that is used on the NetDCU boards, allows even more complex acceptance checking in PeliCan mode, for example using two different but shorter `code`/`mask` pairs. However discussing these is beyond the scope of this document. Please refer to the original CAN controller documentation. Some examples are also given in the NetDCU CANINTF driver documentation.

3.37 struct CanTime

Entries:

`UInt32 low` Low word of the time value

`UInt32 high` High word of the time value

Description:
The CAN bus time in this driver implementation is based on the internal WinCE tick count, i.e. the number of milliseconds since power-on. It is a 64 bit value split in a low and high part with each 32 bits.

3.38 struct CanTransmitData

Entries:

`UInt32` `id`
ID of the message, usually identifying the target device or message type

`byte` `rtr`
Remote transmission request
0: CAN message with data
1: Request data from receiver

`byte` `dlc`
Data length code, i.e. number of message bytes in `msg` (0..8)

`byte[]` `msg`
The bytes of the message itself. Only the first `dlc` bytes are used.

Description:
Data describing a CAN message. Please note that the byte array `msg` can be larger than the message, but only `dlc` bytes are actually transmitted (on send) or valid (on receive).

3.39 struct CanEvent

Entries:

`CanEventFlags` `event_type`
Type of event the structure describes

`CanTime` `time`
Time when this event occurred

`uint` `lost`
Number of messages that were lost since the last `ReadEventData()`

`CanTransmitData` `data`
The message data

`int` `arbitration`
(unused)

Description:
The data of a CAN bus event shows the type of event that occurred, the time when this occurred, and the number of messages that were lost due to slow reading. In case of receive and transmit events, the data of the transferred message is also included. This structure is only used with `ReadEventData()`.

3.40 struct CanProperties

Entries:

`UInt32` `version`
CAN bus driver software version

`string` `name`
Name (description) of the device

`UInt32` `baudmin`
Minimum baud rate supported

`UInt32` `baudmax`
Maximum baud rate supported

`UInt32` `nCommands`
Number of available commands

`CanCommand[]` `commands`
Array with commands. Only the first `nCommands` entries are valid

`UInt32` `nBaudrates`
Number of entries in the predefined baud rate table

`UInt32[]` `baudrates`
Baud rate table. Array with supported baud rates. Only the first `nBaudrates` entries are valid

`CanChipsetFlags` `chipset_flags`
Capabilities of the CAN controller

`UInt32` `nRegisters`
Number of available controller registers

Description:

This structure shows the capabilities of the CAN bus controller and has influence on the value ranges allowed in some of the `CanPort` functions.

- `baudmin`, `baudmax` define the minimum and maximum values for `SetBaudrate()`.
- `nBaudrates` and `baudrates[]` define the possible index values available for `SetBaudrateByIndex()`.
- `nCommands` and `commands[]` define the possible commands that can be used in `SetCommand()`.
- `chipset_flags` shows the capabilities of the controller for `WriteTransmitData()`, `SetCanMode()`, and `SetDefaultFrameFormat()`.
- `nRegisters` shows which registers can be read and written with `ReadRegister()`, `ReadRegisterRM()`, `WriteRegister()`, and `WriteRegisterRM()`.

4 The CanPortException class

The `CanPortException` class defines an exception used in combination with the `CanPort` class. When an error happens within a function of `CanPort`, it throws this kind of exception, so you can react to it in a `try-catch` statement.
The `CanPortException` extends `ApplicationException` by a read-only property `int Reason`, showing the error code why the exception was thrown. This is usually the value returned by the Win32 API via `GetLastWin32Error()`. A typical piece of code would look like this.

```
try
{
    CanPort pCAN = new CanPort("CID1:", ...);
    ...                              //              Use              pCAN
}
catch                         (CanPortException                         e)
{
    switch                                                    (e.Reason)
    {
    case                             CanPort.APIError.ERROR_INVALID_NAME:
        ...  // Handle error
    case                            CanPort.APIError.ERROR_ACCESS_DENIED:
        ...  // Handle error
    }
}
```

When examining the reason, `CanPort.APIError` (see page 30) may be of some help to check for possible error sources.

4.1 CanPortException() (Construction)

Signature 1:
```
CanPortException(string text, int reason)
```

Parameters:

`text`	Error text
`reason`	Error number

Description:
Store given error value as Reason. The error text is automatically completed with "`: Error code <reason>`" where <reason> is the given error number.

Signature 2:
```
CanPortException(string text, int reason,
                 Exception inner)
```
Parameters:

`text`	Error text
`reason`	Error number
`inner`	Inner exception

Description:
Same as above, but with inner exception.

Signature 3:
```
CanPortException(string text)
```

Parameters:

`text`	Error text

Description:
Same as above, but automatically use the result of `GetLastWin32Error()` as error number.

Signature 4:
```
CanPortException(string text, Exception inner)
```

Parameters:

`text`	Error text
`inner`	Inner exception

Description:
Same as above, with inner exception.

Signature 5:
`CanPortException(int reason)`

Parameters:
`reason` Error number

Description:
Use given error number and "`System error`" as error text.

Signature 6:
`CanPortException(int reason, Exception inner)`

Parameters:
`reason` Error number
`inner` Inner exception

Description:
Same as above, but with inner exception.

Signature 7:
`CanPortException()`

Description:
Use `GetLastWin32Error()` as error number and string "`System error`" as error text.

Signature 8:
`CanPortException(Exception inner)`

Parameters:
`inner` Inner exception

Description:
Same as above, but with inner exception.

5 Sample Program CanWrite

This small command line program sends a number of messages over the CAN bus. With each message, the ID is incremented by one. You can select the port, the transfer speed and the number of messages to send on the command line.

Usage:

```
CanWrite        [-?]        [-b        <baudrate]        [-d        <device>]        [-n        <count>]
Send messages over CAN port
Options:
 -?                                                                          Show      this      help
 -b      <baudrate>           Set      line      speed      to      <baudrate>      (default:      1000000)
 -d     <device>                  Set     port     to     use     to     <device>     (default:     CID1:)
 -n <count>     Send <count> messages (default: 1000)
```

Remark:

This program uses the exception error model. When an error occurs inside a `CanPort` function, a `CanPortException` is thrown.

5.1 Source Code

```
using                                                                                                   System;
using                                                                                                   System.Data;
using FS.NetDCU;
namespace                                                                                               MyProgram
{
  class                                                                                                 CanWrite
  {
    static                              int                             Main(string[]                       args)
    {
      string                              devname                             =                           "CID1:";
      uint                              baudrate                              =                           250000;
      uint                                count                               =                             1000;
      bool                              bSetBaudrate                            =                           false;
      int                                                                                               retval;
      uint i;
      Console.WriteLine("CanWrite .NET Version 1.0");
      /*                  Parse                   command                 line                    options                 */
      i                                                       =                                         0;
      while                                                                                         (i<args.Length)
      {
        switch                                                                                          (args[i])
        {
        case      "-?":                                                                     //      Show      usage
        default:
          Console.WriteLine("CanWrite                 [-?]                [-b                 <baudrate]                "
                            +"[-d                     <device>]                       [-n                   <count>]");
          Console.WriteLine("Send                 messages                over                CAN                 port");
          Console.WriteLine();
          Console.WriteLine("Options:");
          Console.WriteLine("     -?                                                                    Show     this     help");
          Console.WriteLine("       -b        <baudrate>                Set        line        speed        to          "
                            +"<baudrate>                              (default:                                 1000000)");
          Console.WriteLine("       -d      <device>                        Set      port      to      use      to        "
                            +"<device>                                (default:                                  CID1:)");
          Console.WriteLine("        -n        <count>                                                 Send        <count>        "
                            +"messages (default: 1000)");
          return 0;
        case      "-d":                                                                     //      Parse      device
          if                                  (i+1                                >=                          args.Length)
            goto                                                                                          default;
          devname                                                 =                                       args[++i];
          break;
        case      "-b":                                                                     //      Parse      baudrate
          if                                  (i+1                                >=                          args.Length)
            goto                                                                                          default;
          bSetBaudrate                                                  =                                   true;
          baudrate                                          =                               uint.Parse(args[++i]);
          break;
        case      "-n":                                                                 //      Parse      package      count
          if                                  (i+1                                >=                          args.Length)
            goto                                                                                          default;
          count                                             =                               uint.Parse(args[++i]);
          break;
        }
        i++;
      }
      /*                                Send                                messages                                */
      try
      {
        CanPort.CanTime                                                                                 time;
        CanPort.CanProperties prop;
        /*                  Open                and                 configure                   port                */
        CanPort pCAN = new CanPort(devname, CanPort.CanAccess.WRITE);
        if                                                                                              (bSetBaudrate)
          pCAN.SetBaudrate(baudrate);
        pCAN.GetBaudrate(out baudrate);
        /*                Read                and                 print                   properties                */
        pCAN.ReadProperties(out                                                                         prop);
        Console.WriteLine();
        Console.WriteLine("CAN                                          Bus                                     Properties:");
        Console.WriteLine("Device             name:                                             '{0}'",           prop.name);
        Console.WriteLine("Version:                                                               {0}",         prop.version);
        Console.WriteLine("Baudrate:                                      min:      {0}      Hz,      max:      {1}      Hz",
                          prop.baudmin,                                                                         prop.baudmax);
        Console.Write("{0}             Commands:                                                ",              prop.nCommands);
        for             (i             =             0;            i            <            prop.nCommands;            i++)
          Console.Write("                                       0x{0:X2}",                                prop.commands[i]);
        Console.WriteLine();
        Console.Write("{0}                Baudrates:                                        ",                  prop.nBaudrates);
        for             (i             =             0;            i            <           prop.nBaudrates;            i++)
          Console.Write("                                     {0}",                                       prop.baudrates[i]);
        Console.WriteLine();
        Console.WriteLine("Chipset                                        flags:                                    0x{0:X}",
                          prop.chipset_flags);
        Console.WriteLine("{0}                                  Registers",                                 prop.nRegisters);
        Console.WriteLine();
        pCAN.ReadTime(out                                                                                       time);
        Console.WriteLine("Current                            time:                                           {0}:{1}",
                          time.high,                                                                          time.low);
        Console.WriteLine();
        /*                              Transmit                                  messages                              */
        Console.WriteLine("Start          sending          {0}          messages          to          {1}          with          "
                          +"baudrate           {2}           Hz",           count,           devname,           baudrate);
        for                               (i=1;                                 i<count;                                i++)
        {
          byte[] msg = new byte[8] {1, 2, 3, 4, 5, 6, 7, 8};
          pCAN.WriteTransmitData(CanPort.CanTransmitFormat.DEFAULT,
                                 i,                                         0,                                      msg);
        }
      }
      catch                                         (CanPortException                                       e)
      {
```

```
        /*                      Print                       error                       message                     */
        Console.WriteLine(e.Message);
        return                                                                                                      1;
      }
      return                                                                                                        0;
    }                                                 //                                                      Main()
  }                                  //                                  class                                CanWrite
} // namespace MyProgram
```

Program variant:

The above version uses `WriteTransmitData()`, which is the binary interface to the CAN driver.

```
pCAN.WriteTransmitData(CanPort.CanTransmitFormat.DEFAULT,
                       i,                                   0,                                  msg);
```

The following line in the transfer loop would use the text interface with `Write()` instead.

```
pCAN.Write(i.ToString("X") + " 0 8 1 2 3 4 5 6 7 8\n");
```

6 Sample Program CanRead

This small command line program receives a number of messages over the CAN bus and prints its content to the console. On the command line you can select the port, the transfer speed and the number of messages to receive until the program terminates.

Usage:

```
CanRead         [-?]         [-b         <baudrate]         [-d         <device>]         [-n         <count>]
Read messages from CAN port
Options:
 -?                                                                                  Show      this      help
 -b      <baudrate>            Set      line      speed      to      <baudrate>      (default:      1000000)
 -d     <device>                   Set     port     to     use     to     <device>     (default:     CID1:)
 -n <count>     Stop after <count> messages (default: 1000)
```

Remark:

This program uses the C style return code error model. When an error occurs inside a `CanPort` function, the function returns an error code. Otherwise it returns 0.

6.1 Source Code

```
using System;
using System.Data;
using FS.NetDCU;
namespace MyProgram
{
  class CanRead
  {
    static int Main(string[] args)
    {
      string devname = "CID1:";
      uint baudrate = 250000;
      uint count = 1000;
      bool bSetBaudrate = false;
      int retval;
      uint i;
      Console.WriteLine("CanRead .NET Version 1.0");
      /* Parse command line options */
      i = 0;
      while (i<args.Length)
      {
        switch (args[i])
        {
        case "-?":
        default:
          Console.WriteLine("CanRead [-?] [-b <baudrate]"
                            +"[-d <device>] [-n <count>]");
          Console.WriteLine("Read messages from CAN port");
          Console.WriteLine();
          Console.WriteLine("Options:");
          Console.WriteLine("     -? Show this help");
          Console.WriteLine("       -b <baudrate> Set line speed to "
                            +"<baudrate> (default: 1000000)");
          Console.WriteLine("       -d <device> Set port to use to "
                            +"<device> (default: CID1:)");
          Console.WriteLine("       -n <count> Stop after <count> "
                            +"messages (default: 1000)");
          return 0;
        case "-d":
          if (i+1 >= args.Length)
            goto default;
          devname = args[++i];
          break;
        case "-b":
          if (i+1 >= args.Length)
            goto default;
          bSetBaudrate = true;
          baudrate = uint.Parse(args[++i]);
          break;
        case "-n":
          if (i+1 >= args.Length)
            goto default;
          count = uint.Parse(args[++i]);
          break;
        }
        i++;
      }
      /* Read messages */
      try
      {
        CanPort.CanAcceptanceFilter filter;
        CanPort.CanEventFlags mask;
        /* Open and configure port */
        CanPort pCAN = new CanPort(devname, CanPort.CanAccess.READ);
        pCAN.HandleErrorsViaReturn(true); // Handle errors ourselves
        if (bSetBaudrate)
        {
            retval = pCAN.SetBaudrate(baudrate);
            if (retval != 0)
                throw new CanPortException("Can't set baudrate",
                                           retval);
        }
        retval = pCAN.GetBaudrate(out baudrate);
        if (retval != 0)
            throw new CanPortException("Can't get baudrate", retval);

        /* Read and print some configuration */
        Console.WriteLine();
        retval = pCAN.ReadAcceptanceFilter(out filter);
        if (retval != 0)
            throw new CanPortException("Can't read acceptance filter",
                                       retval);
        Console.WriteLine("Previous acceptance filter: code=0x{0:X}, "
                          +"mask=0x{1:X}", filter.code, filter.mask);
        filter.code = 0;
        filter.mask = 0xFFFFFFFF;
        retval = pCAN.WriteAcceptanceFilter(ref filter);
        if (retval != 0)
            throw new CanPortException("Can't write new acceptance "
                                       +"filter", retval);
        retval = pCAN.ReadAcceptanceFilter(out filter);
        if (retval != 0)
            throw new CanPortException("Can't read acceptance filter",
                                       retval);
        Console.WriteLine("New acceptance filter: code=0x{0:X}, "
                          +"mask=0x{1:X}", filter.code, filter.mask);
        retval = pCAN.GetCommMask(out mask);
        if (retval != 0)
            throw new CanPortException("Can't get comm mask", retval);
        Console.WriteLine("Previous event mask: {0}", mask);
        mask = CanPort.CanEventFlags.CANBUS_ALL;
        retval = pCAN.SetCommMask(mask);
        if (retval != 0)
```

```
        throw new CanPortException("Can't set new mask", retval);
      retval = pCAN.GetCommMask(out mask);
      if (retval != 0)
        throw new CanPortException("Can't get comm mask", retval);
      Console.WriteLine("New event mask: {0}", mask);
      /* Receive messages */
      Console.WriteLine("Start receiving {0} messages on {1} with "
                 +"baudrate {2} Hz", count, devname, baudrate);
      // >>>>>>>>>>>>>>>>>>>>
      while (true)
      {
        CanPort.CanEvent ev;
        /* Read events until error */
        while (pCAN.ReadEventData(out ev) == 0)
        {
          /* Print CAN bus time when event occurred */
          Console.Write("{0:X8}:{1:X8}: ",
                     ev.time.high, ev.time.low);
          switch (ev.event_type)
          {
          case CanPort.CanEventFlags.RECEIVED:
            Console.Write("Received: id=0x{0:X8}, rtr={1}, "
                       +"dlc={2}, lost={3}", ev.data.id,
                       ev.data.rtr, ev.data.dlc, ev.lost);
            if (ev.data.dlc > 0)
            {
                Console.WriteLine();
                Console.Write(" msg:");
                for (i = 0; i < ev.data.dlc; i++)
                    Console.Write(" 0x{0:X2}", ev.data.msg[i]);
            }
            Console.WriteLine();
            break;
          case CanPort.CanEventFlags.TRANSMITTED:
            Console.Write("Transmitted: id=0x{0:X8}, rtr={1}, "
                       +"dlc={2}", ev.data.id, ev.data.rtr,
                       ev.data.dlc, ev.lost);
            if (ev.data.dlc > 0)
            {
                Console.Write(", msg:");
                for (i = 0; i < ev.data.dlc; i++)
                    Console.Write(" 0x{0:X2}", ev.data.msg[i]);
            }
            Console.WriteLine();
            break;
          case CanPort.CanEventFlags.WARNING:
            Console.WriteLine("Warning");
            break;
          case CanPort.CanEventFlags.BUS_ERROR:
            Console.WriteLine("Error");
            break;
          case CanPort.CanEventFlags.ARBITRATION_LOST:
            Console.WriteLine("Arbitration lost");
            break;
          case CanPort.CanEventFlags.OVERRUN:
            Console.WriteLine("Overrun");
            break;
          case CanPort.CanEventFlags.DEVICE_CHANGED:
            Console.WriteLine("Device changed");
            break;
          case CanPort.CanEventFlags.PASSIVE:
            Console.WriteLine("Passive");
            break;
          case CanPort.CanEventFlags.ENTERING_STANDBY:
            Console.WriteLine("Entering standby");
            break;
          case CanPort.CanEventFlags.LEAVING_STANDBY:
            Console.WriteLine("Leaving standby");
            break;
          }
          if (--count == 0)
            goto DONE;
        }
        /* Wait for next event */
        retval = pCAN.WaitCommEvent(out mask);
        if (retval != 0)
            throw new CanPortException("Error in WaitCommEvent()",
                           retval);
      }
    DONE:
      ;
      // <<<<<<<<<<<<<<<<<<<<
    }
    catch (CanPortException e)
    {
      /* Print error message */
      Console.WriteLine(e.Message);
      return 1;
    }
    return 0;
  } // Main()
 } // class CanRead
} // namespace MyProgram
```

Program variant:

The above version uses `ReadEventDataData()`, which is the binary interface to the CAN driver. The program shows how the different events can be interpreted, and instead of printing the content to the console, the program could have used the data in any other way.

To have the program use the text interface with `Read()`, the part between
`// >>>>>>>>>>>>>>>>>>>>`
and
`// <<<<<<<<<<<<<<<<<<<<`
needs to be replaced with the following code fragment.

```
while                                                                                                   (count--)
{
  string message;
  retval                                =                                   pCAN.Read(out                                 message);
  if                                      (retval                                       !=                                        0)
    throw                    new                  CanPortException("Read                    error",                    retval);
  Console.WriteLine(message);
}
```

This looks much easier. But it is only suited when the format provided by `Read()` is exactly what you want. To react differently according to each event type, it is definitely more effort to parse the text output from `Read()` than using the binary interface with `ReadEventData()` directly.

7 Appendix

Important Notice

The information in this publication has been carefully checked and is believed to be entirely accurate at the time of publication. F&S Elektronik Systeme assumes no responsibility, however, for possible errors or omissions, or for any consequences resulting from the use of the information contained in this documentation.
F&S Elektronik Systeme reserves the right to make changes in its products or product specifications or product documentation with the intent to improve function or design at any time and without notice and is not required to update this documentation to reflect such changes.
F&S Elektronik Systeme makes no warranty or guarantee regarding the suitability of its products for any particular purpose, nor does F&S Elektronik Systeme assume any liability arising out of the documentation or use of any product and specifically disclaims any and all liability, including without limitation any consequential or incidental damages.
Specific testing of all parameters of each device is not necessarily performed unless required by law or regulation.
Products are not designed, intended, or authorized for use as components in systems intended for applications intended to support or sustain life, or for any other application in which the failure of the product from F&S Elektronik Systeme could create a situation where personal injury or death may occur. Should the Buyer purchase or use a F&S Elektronik Systeme product for any such unintended or unauthorized application, the Buyer shall indemnify and hold F&S Elektronik Systeme and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, expenses, and reasonable attorney fees arising out of, either directly or indirectly, any claim of personal injury or death that may be associated with such unintended or unauthorized use, even if such claim alleges that F&S Elektronik Systeme was negligent regarding the design or manufacture of said product.
Specifications are subject to change without notice.

Warranty Terms

Hardware Warranties
F&S guarantees hardware products against defects in workmanship and material for a period of one (2) year from the date of shipment. Your sole remedy and F&S's sole liability shall be for F&S, at its sole discretion, to either repair or replace the defective hardware product at no charge or to refund the purchase price. Shipment costs in both directions are the responsibility of the customer. This warranty is void if the hardware product has been altered or damaged by accident, misuse or abuse.

Software Warranties
Software is provided "AS IS". F&S makes no warranties, either express or implied, with regard to the software object code or software source code either or with respect to any third party materials or intellectual property obtained from third parties. F&S makes no warranty that the software is useable or fit for any particular purpose. This warranty replaces all other warranties written or unwritten. F&S expressly disclaims any such warranties. In no case shall F&S be liable for any consequential damages.

Disclaimer of Warranty

THIS WARRANTY IS MADE IN PLACE OF ANY OTHER WARRANTY, WHETHER EXPRESSED, OR IMPLIED, OF MERCHANTABILITY, FITNESS FOR A SPECIFIC PURPOSE, NON-INFRINGEMENT OR THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION, EXCEPT THE WARRANTY EXPRESSLY STATED HEREIN. THE REMEDIES SET FORTH HEREIN SHALL BE THE SOLE AND EXCLUSIVE REMEDIES OF ANY PURCHASER WITH RESPECT TO ANY DEFECTIVE PRODUCT.

Limitation on Liability

UNDER NO CIRCUMSTANCES SHALL F&S BE LIABLE FOR ANY LOSS, DAMAGE OR EXPENSE SUFFERED OR INCURRED WITH RESPECT TO ANY DEFECTIVE PRODUCT. IN NO EVENT SHALL F&S BE LIABLE FOR ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES THAT YOU MAY SUFFER DIRECTLY OR INDIRECTLY FROM USE OF ANY PRODUCT. BY ORDERING THE PRODUCT, THE CUSTOMER APPROVES THAT THE F&S PRODUCT, HARDWARE AND SOFTWARE, WAS THOROUGHLY TESTED AND HAS MET THE CUSTOMER'S REQUIREMETS AND SPECIFICATIONS