Software Documentation

F&S CAN Driver

Version 2.06
2015-03-23

© F&S Elektronik Systeme GmbH
Untere Waldplätze 23
D-70569 Stuttgart
Fon: +49(0)711-123722-0
Fax: +49(0)711 – 123722-99

Introduction

Overview

The Controller Area Network (CAN) is a two wire bus with differential signals to communicate between different kinds of sensors, controllers and other devices. It is often used in automotive and industrial applications. The transfer speed can be up to 1 Mbit/s.
Boards from F&S offer one or two CAN ports for direct integration of the device into a CAN network.
This documentation describes the functionality of the CANDRV driver, how to install this driver and how to use it in own software applications. Please check if the driver is installed and running. You can do this by opening the tool remote registry editor and search the entries in `[HKLM\Drivers\Active]`. If you don’t’ find the driver, you have to install it for each CAN interface.
We provide a collection of small CAN sample programs called `CanTestSuite` to show the usage of the driver and to help configuring and testing your applications. The source code of these programs is also available (see appendix).

Table of Content

1 CAN Bus Operation

Data can be sent in short messages across the CAN bus. These messages are packed into data frames. A data frame consists of

- a network-wide unique identifier (ID), usually identifying a function of the affected device,
- a Remote Transmission Request flag (RTR), set when requesting information from a remote device,
- a Data Length Code (DLC), telling the length of the message data (0..8),
- and up to eight message data bytes (MSG[0..7]).

There are additional data bits for synchronisation, error checking (CRC) and acknowledgement, but these are of no big importance when only using the CAN driver. They are all handled within the hardware controller or the software driver. Therefore we'll only present a simplified view of the frames.

Frame Types
There exist two different types of frames: Standard Frames and Extended Frames. Standard Frames have an ID of 11 bit length. CAN networks using only Standard Frames are conforming to the *CAN2.0A* standard (also called BasiCan).

Standard Frame:

11-bit ID	RTR	DLC	MSG[0]	...	MSG[7]

Extended Frames have an ID of 29 bit length. Any network that is capable of using Extended Frames must also be capable of accepting Standard Frames, in any random order. Therefore networks using Standard *and* Extended Frames are conforming to the *CAN2.0B* standard (also called PeliCan).

Extended Frame:

29-bit ID	RTR	DLC	MSG[0]	...	MSG[7]

A device can start sending a frame when the bus is idle. If another device is starting to send at the same time, this is recognised at the first different bit. As bits are recessive (1) or dominant (0) on the bus, the dominant bit wins. This usually happens while transmitting the ID. Therefore the device with the lower ID (=higher priority) wins. This device can continue to send.
The other device must immediately stop sending. It is said to have lost bus arbitration. This device can try to send again when the current frame is finished. If no device with higher priority sends at the same time anymore, it can eventually transmit its complete frame.

CAN Identifiers
Which CAN identifier is used for which purpose is basically up to the designer of the CAN network. The only restriction is that IDs must be unique. However there exist different standards like *DeviceNet* and *CanOpen* that have predefined mappings of device types and functions to IDs. Please refer to these standards when using the CAN driver in a network together with such devices.
There is a *CanOpen* stack available for the F&S CAN driver from a third party company. Please contact F&S if you are interested in using it.

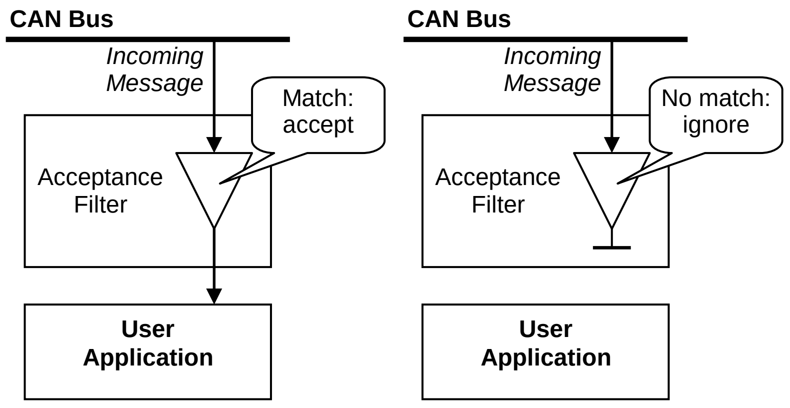
Remote Transmission Request (RTR)
Usually the CAN ID identifies some kind of function of a specific device. For example a sensor can regularly report its measured value with a specific ID. Or a controller can switch an actor with a specific function on or off by sending a message with this special function ID.
Sometimes one device needs information from another device at a specific point of time, for example if a sensor does not report its value automatically but must be polled. This is done by a request frame. The controlling device sends a CAN frame with the function ID of the

device to poll and the RTR bit set. The polled device then may respond directly within the same frame (overruling the recessive RTR bit of the original sender, taking over the transmission and directly sending the data), or in a separate data frame later.

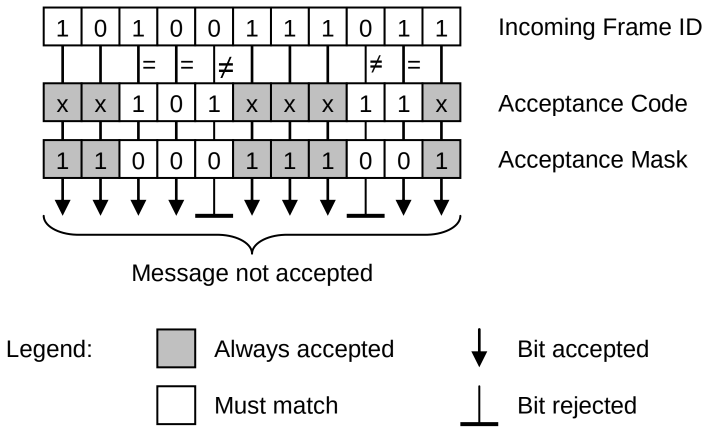
Data Length Code (DLC) And Data Bytes
The data length code defines how many data bytes are appended to the frame. The shortest frame has no data bytes (DLC=0) and the longest data frame has eight data bytes (DLC=8).
Remark: As DLC is 4 bits in the CAN frame, it is theoretically possible to set it to a value higher than 8. This is discouraged by the CAN standard and some CAN devices my even interpret this as an error. Nonetheless this CAN driver will transfer DLC as it is set, i.e. 0..15. But all values higher than 8 will still only transport eight data bytes.

2 Acceptance Filters

All devices connected to the CAN bus basically can see and receive any message. However usually only a few messages are relevant for a specific device. Therefore it is possible to restrict the reception of messages to specific message IDs. This may be a single ID, a range of IDs, or even a group of different IDs or ID ranges. This process is called *acceptance filtering*. Only those messages that match the given filter IDs are received, all other messages on the CAN bus are ignored.



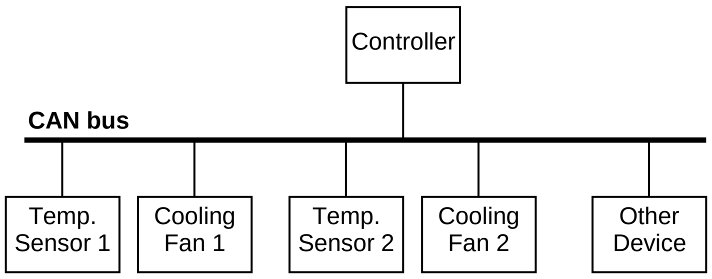
An Acceptance Filter consists of a *code* and a *mask*. The acceptance code determines the ID to match. The acceptance mask tells which bits of the received frame ID are relevant and must match exactly to the acceptance code to be accepted and which bits of the ID don't matter and are always accepted.



Normally only the ID of the incoming frame is checked by the acceptance filter. But there is an exception. Most CAN controllers have a special feature when comparing Standard Frame IDs: here they can extend the acceptance test to the first two data bytes. Then not only the 11-bit ID is tested, but also up to 16 bits of the first two data bytes.
This feature is supported by this CAN driver. But please note that this is only possible with Standard Frames. When receiving Extended Frames, the test can only check the 29 bits of the extended ID and no additional data bytes.

3 CAN Bus Network Example

Let’s consider a larger machine, which can get hot in some places. There are cooling fans installed, but they should only run selectively if the specific machine part is getting too hot. Therefore the cooling fans are connected together with a set of temperature sensors and a controlling unit in a CAN network. And there are also some other devices present.



Let’s assume that the temperature value of a sensor can be coded in eight bits and will be sent every second. And the signal of switching a cooling fan on or off is coded in one bit and is only sent when required. Let’s assume further that a temperature sensor has an ID of 0x100 and a cooling fan of 0x200.
Then the network could work like this.
A temperature sensor sends every second a frame of the following type:

ID=0x100	RTR=0	DLC=2	Sensor Number	Temperature

It does not need to react to any incoming messages.

The controller accepts all messages with ID 0x100. Therefore it will have an acceptance filter with code = 0x100 and a mask that requires all bits to match the ID.
It will look into the first data byte to determine the number of the temperature sensor and depending on the new temperature in the second data byte it will probably send a message to the corresponding cooling fan to switch on or off.

ID=0x200	RTR=0	DLC=2	Fan Number	0=off/1=on

A cooling fan does not send any message, but reacts to messages with ID=0x200. However Fan 1 is only interested in those messages directed to Fan 1, not in messages for other fan numbers. Fan 2 is only interested in messages for Fan 2, and so on. This shows that it makes sense to have the possibility to extend the acceptance filter to the first data byte. A cooling fan best configures its acceptance mask to match ID and first data byte and the acceptance code to 0x200 plus the local fan number in the data section.

Remark
When only a few different device types are in a CAN network, the IDs themselves can be used to transmit data. This makes transmission extremely efficient.
For example if the above network only has at most four temperature sensors, both the sensor number and the temperature could be coded in 10 bits within the ID itself, leaving the last bit of the ID as marker for temperature sensor or other device. Then the cooling fans could be coded as a group of the remaining device IDs and there would still be room for other devices. Here is a coding that would work more efficiently. However it is not as flexible anymore as the first solution.

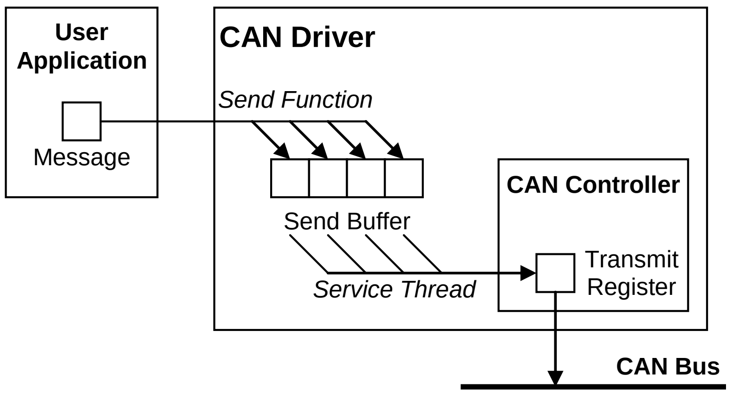
ID (binary)	Meaning
`1 nn tttttttt`	Report temperature `t` (0..255) of sensor `n` (0..3)
`011 11111 nn s`	Switch cooling fan `n` (0..3) to state `s` (on=1, off=0)
`0xx xxxxx yyy, xxxxxxx!=1111111`	Messages from any other devices

If identifier `1 10 01010000` would be received on the CAN bus, this would mean that temperature sensor 2 reports the current temperature value 0x50. If the ID `011 11111 01 0` is received on the CAN bus, this would mean that cooling fan 1 should be switched off. If ID `001 00101 100` is received on the bus, this is the message of some other device on the bus.
This bus configuration shows the need for more complex acceptance filters.

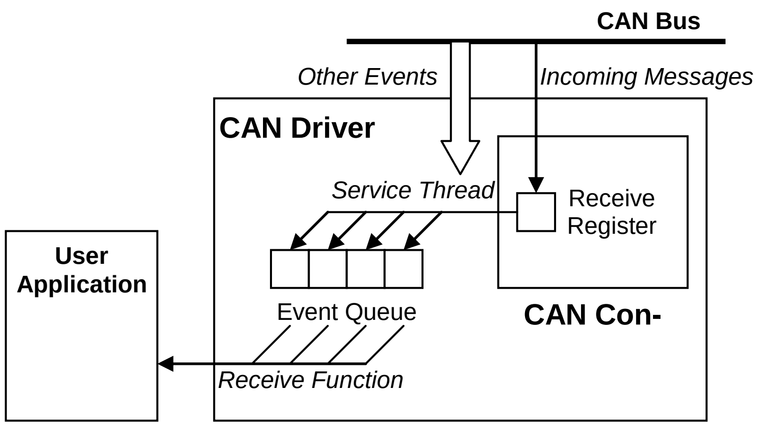
- The controller is interested in any message coming from any temperature sensor. Therefore it would set its acceptance filter to only check the most significant ID bit to be `1` and accept all other ID bits. By looking at the actually received ID, it could determine which sensor sent which temperature.
  Acceptance mask:`011 11111111` (0=check, 1=accept)
  Acceptance code: `1xx xxxxxxxx` (`x`=don't care)
- The cooling fans are interested in the switching messages. Therefore they would set their acceptance filters to check the eight most significant bits to be exactly `01111111`, the next two bits to be the own fan number and to accept the last bit no matter what it is. They would then analyse this last bit of the actually received ID and switch the fan on or off. Here e.g. the settings for Fan 1:
  Acceptance mask:`000 00000001` (0=check, 1=accept)
  Acceptance code: `011 1111101x` (`x`=don't care)

4 Send Buffer and Event Queue

Transmission on the CAN bus takes some time. Therefore data that is to be sent on the CAN bus usually is not sent directly, but stored in the so-called *Send Buffer*. This allows the sending function to return immediately.
Then the CAN service routine, a separate background thread of the driver, is responsible for actually transmitting these entries of the send buffer on the CAN bus as fast as possible. Every time a message is completed by the controller hardware, it takes the next message from the send buffer and serves it to the controller to be transmitted.



On the other side everything that happens on the CAN bus is handled as an event. An incoming message for example is one type of event. Some error that happened on the bus is another type of event. And there are more types of events. These events are stored in an event queue within the driver and the user application can react to them.
It is important to understand that there is no separate receive buffer reserved only for received messages. All incoming messages are stored in the standard event queue together with all other kinds of possible CAN bus events.



When reading from the CAN device, this is actually fetching the next event from the event queue.

5 Possible interface conflicts

On some modules there is a mutual interference between CAN and SPI interface. This is caused by the fact that the CAN interface internally is connected to the same SPI controller that is used for the external so called (Native)SPI interface. This might lead into malfunction as both drivers would try to access the SPI bus simultaneously.
To overcome this restriction, the “DirectInterface” feature of the CAN driver must be disabled. In doing so the CAN driver will also use the available SPI driver to access CAN controller.
Here is a list of modules that are affected by this mutual conflict:

Board
PicoCOM1
PicoCOM2
PicoCOM4

For details how to set up the proper configuration, please refer to the platform dependent readme file that comes with the CAN driver package. Or contact support@fs-net.de for support.
Note:
Please note that disabling the DirectInterface is **only** required if CAN interface and (Native)SPI interface are used simultaneously.

6 Installing the CAN Software Driver

The CANDRV driver is usually already pre-installed on the board as `CID1:`. If not, there may be two different possibilities for installation.

6.1 Installation with the CAB File

If you have the Windows Cabinet File `candrv.cab` available, just copy this file to the board (e.g. to the root directory). Then open the command line and type the following command:

```
wceload /noui candrv.cab
```

This will install the CAN driver as `CID1:` with the default settings. If you need other settings, you can edit the registry values anytime after installation is complete.

6.2 Manual Installation

You can also do the installation by hand. This requires setting some registry entries. Installation of the CAN driver takes place in the registry under
`[HKLM\Drivers\BuiltIn\CID1]`

Entry	Type	Value	Description
`Dll`	String	`can-drv.dll`	Driver DLL
`FriendlyName`	String	`PicoCOM CAN driver`	Description
`Prefix`	String	`CID`	For `CID1:`
`Index`	DWORD	`1`	For `CID1:`
`Order`	DWORD	`20`	Load sequence
`DeviceArrayIndex`	DWORD	`0`	Use CAN port device 0
`Ioctl`	DWORD	`4`	Call post initialisation function
`TxMode`	DWORD	`7`	Transmit mode(*)
`SendBufferSize`	DWORD	`100`	Maximum number of messages in Send Buffer (*)
`EventQueueSize`	DWORD	`200`	Maximum number of events in Event Queue
`Priority256`	DWORD	`103`	Thread priority
`Debug`	DWORD	`0`	Debug verbosity
`Baudrate`	DWORD	`1000000`	Default baud rate
`CanMode2B`	DWORD	`0`	Default CAN bus mode
`Format`	DWORD	`0`	Default frame format
`Virtualize`	DWORD	`0`	Virtual CAN bus mode
`AcceptanceCode`	DWORD	`0`	Code value for default acceptance filter

			(*)
`AcceptanceMask`	DWORD	`0`	Mask value for default acceptance filter (*)
`MaskActive`	DWORD	`0`	Acceptance filter mask logic (*)
`Align`	DWORD	`0`	ID and acceptance filter alignment (*)
`IRQ`	DWORD	`30`	Default IRQ for CAN controller
`DirectInterface`	DWORD	`0`	How to access SPI port (*)(**)
`SPIDevice`	String	`SPI3:`	Device name of the SPI port (**)

(*) Registry entry available since V2.x
(**) Registry entry only available on PicoCOM1/PicoCOM2.
Most of the values will get meaningful defaults if omitted, only those values highlighted in grey above really have to be given. The library `candrv.dll` has to be stored in flash memory into the `\FFSDISK` directory, if it is not already pre-loaded in the kernel.

6.3 Detailed Description of the Registry Values

`Dll`
Name of the CAN driver library, usually `CANDRV.DLL`. If the driver is not loaded, try using the full path to the library here.

`FriendlyName`
Short description of the driver function.

`Prefix`
Three upper case characters used for the device name. Usually `CID` for “CAN Interface Driver”.

`Index`
The number of the device name. This has to be a number from 0 to 9. Usually the first (or only) device gets number 1.

`Order`
All device drivers are started by the Device Manager. This value decides in what sequence the drivers are loaded: the higher the number the later they are loaded.

`IoCtl`
When this entry is set, the Device Manager will automatically open the device once after the driver is loaded and calls function `DeviceIoControl()` with this number as argument. Then the device is closed again.
`DeviceArrayIndex`
This zero based index tells the driver which CAN port to use by this driver instance. The PicoCOM1, PicoCOM2 and PicoMOD3 only support one CAN port, therefore this entry `DeviceArrayIndex` must be set to `0`.

`TxMode`
This entry sets the transmission mode. Every bit of `TxMode` controls a different setting. See page 24 for more details.

Bit	Value	Meaning
0	0	*Single message mode*: a message must be transmitted before the send function returns; no send buffer is used
	1	*Send buffer mode*: a message is placed into the send buffer and the send function returns immediately
1	0	Don’t use `CANBUS_EVENT_TRANSMITTED`
	1	Generate `CANBUS_EVENT_TRANSMITTED` when a message was sent
2	0	A message is considered as sent when it is taken from the send buffer and placed into the hardware transmit register of the controller.
	1	A message is considered as sent when it was actually transmitted by the controller.

Table 1: Registry value - TxMode

`UseTxIRQ` (obsolete)
V1.x of the CAN driver used a registry entry `UseTxIRQ` which could be `0` or `1`. This entry is still recognised by the V2.x driver to not break compatibility, but `TxMode` is preferred now. Setting `UseTxIRQ=0` is the same as setting `TxMode=0`, setting `UseTxIRQ=1` is the same as setting `TxMode=7`.

`SendBufferSize`
If *Send Buffer Mode* is activated in `TxMode`, the driver uses a send buffer for transmitting messages. `SendBufferSize` defines how many messages should fit into that buffer.
If *Single Message Mode* is activated in `TxMode`, no send buffer is used and `SendBuffer-Size` is ignored.

`EventQueueSize`
Every action happening on the CAN bus may generate an event that is stored in the event queue. `EventQueueSize` defines how many incoming events have room in the event queue. If events arrive when the queue is full, these events are lost.

`Priority256`
The driver has a background service thread, taking messages from the send buffer and putting them to the CAN controller or taking messages and events from the CAN controller and putting them into the event queue.
`Priority256` defines the Windows CE thread priority for this service thread. A lower number means higher priority, a higher number means lower priority. Usually Windows CE device drivers use priority values around `100`.

`Debug`
By setting `Debug` to a value other than zero, the driver will print additional diagnostics messages to the serial debug port to help analysing any errors. Please note that this might slow down the driver performance significantly. If not needed. keep this value at `0`.

`Baudrate`
The `Baudrate` entry tells the driver which baud rate to use on the CAN bus (in Hz or bits/s). The baud rate can also be modified at runtime with the functions `IOCTL_CAN_SET_BAUDRATE_BY_CONSTANT` and `IOCTL_CAN_SET_BAUDRATE`.

`CanMode2B`

This entry sets the default operation mode of the CAN controller, CAN2.0A or CAN2.0B. This has influence on the frame format that can be used.
The operation mode can also be modified at runtime with the function `IOCTL_CAN_SET_CAN_MODE`.

`CanMode2B`	Meaning
0	`CANBUS_FORMAT_CAN_2_0_A`: Use Can2.0A mode (Standard Frames only)
1	`CANBUS_FORMAT_CAN_2_0_B`: Use Can2.0B mode (Standard and/or Extended Frames)

Table 2: Registry value – CanMode2B

`Format`

Set the default frame format to be used when sending and receiving messages. This can also be modified at runtime with the function `IOCTL_CAN_SET_DEFAULT_FRAME_FORMAT`.

`Format`	Meaning
0	`CANBUS_TRANS_FMT_DEFAULT` Use the default format for this mode: `CanMode2B=0`: Standard Frames `CanMode2B=1`: Extended Frames
1	`CANBUS_TRANS_FMT_STD` Use Standard Frames
2	`CANBUS_TRANS_FMT_EXT` Use Extended Frames

Table 3: Registry value - Format

`Virtualize`

Entry `Virtualize` decides whether one file handle “sees” the messages transmitted by another file handle or not. For a detailed description of Virtualize Mode see also page 66.

Virtualize	Meaning
0	No virtual CAN bus. A message that is sent can only be received by other devices. No other open file handle on this board will receive this message, even if the acceptance filter matches.
1	Use virtual CAN bus. Every other open file handle on this board with matching acceptance filter will receive a sent message as if it had actually been transmitted over the CAN bus.

Table 4: Registry value - Virtualize

In any case only the file handle that sent the message will get an event `CANBUS_EVENT_TRANSMITTED`.
This mode can also be switched at runtime with the function `IOCTL_CAN_SET_COMMAND`.using the command values `CANBUS_CMD_VIRTUALIZE_ON` to switch the mode to 1 and `CANBUS_CMD_VIRTUALIZE_OFF` to switch the mode to 0.

AcceptanceCode
This is the default setting for the `code` part of the acceptance filter. The code is compared to every received message ID and, depending on the acceptance mask, decides whether the incoming message is accepted or ignored.
The acceptance code can be modified at runtime with the function `IOCTL_CAN_WRITE_ACCEPTANCE_FILTER`.

AcceptanceMask
This is the default setting for the `mask` part of the acceptance filter. The mask decides which bits of the incoming message ID must match the acceptance code and which bits are always accepted.
The acceptance mask can be modified at runtime with the function `IOCTL_CAN_WRITE_ACCEPTANCE_FILTER`.

MaskActive
The `MaskActive` entry defines the logic of the acceptance mask, i.e. which bits of the incoming message ID are always accepted and which bits require a match to the acceptance code.

`MaskActive` setting	Acceptance Mask bit	ID bit of arriving message...
0	0	...must match acceptance code bit
	1	...is always accepted
1	0	...is always accepted
	1	...must match acceptance code bit

Table 5: Registry value - MaskActive

Align
CAN message IDs, acceptance codes and acceptance masks are passed to the driver as unsigned 32-bit values. Depending on the application, it might be useful to have them all aligned left (starting at bit 31), aligned at bit 28 (to make it similar to Extended Frames), aligned at bit 10 (to make it similar to Standard Frames), or each aligned to its individual size. This can be configured with the `Align` entry.

`Align`	Standard-Frame-ID	Standard-Frame-Filter (Code & Mask)	Extended-Frame-ID	Extended-Frame-Filter
0	Bits 10..0	ID: Bits 31..21 Data: Bits 15..0	Bits 28..0	ID: Bits 31..3 no Data
1	Bits 10..0	ID: Bits 10..0 no Data	Bits 10..0	ID: Bits 10..0 no Data
2	Bits 28..18	ID: Bits 28..18 Data: Bits 15..0	Bits 28..0	ID: Bits 28..0 no Data
3	Bits 31..21	ID: Bits 31..21 Data: Bits 15..0	Bits 31..3	ID: Bits 31..3 no Data

Table 6: Registry value - Align

If you used V1.x of the driver and want the same behavior in V2.x, then use the following setting, depending on the CAN mode you used (registry entry `CanMode2B`).

`CanMode2B`	`Align`
0	1
1	0

`IRQ`

This entry tells the CAN driver on which interrupt source the CAN controller will signal its interrupts. This is hardware dependent and usually does not require any modification by the user.

DirectInterface

The CAN controller on the PicoCOM1 and PicoCOM2 is accessed via SPI. Starting with V2.4, the driver can be used together with the SPI driver. This requires that the CAN driver accesses the SPI port via the standard SPI driver. The previous method, where the CAN driver directly accessed the SPI port is still available as Direct Interface.

`DirectInterface`	Meaning
0	Use SPI driver with device name given in `SPIDevice`; this allows using CAN and SPI driver in parallel.
1	Directly access the SPI port; this setting is only allowed if no SPI driver is used on the SPI1 bus.

Table 7: Registry value - DirectInterface

SPIDevice

This entry defines the name of the SPI device to use when `DirectInterface` is set to `0`.

7 Using the CAN Driver in Applications

7.1 Driver Interface

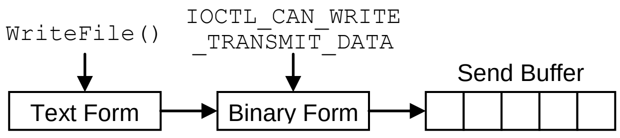
All CAN driver related values and data structures are defined in the header file `canbu-sio.h`. This file must be included in any program using the CAN driver interface. It is listed in Appendix A (page 74).
The driver uses the standard Stream Driver Interface. That means access is done by opening a device file (usually `CID1:`) and then using functions like `WriteFile()` and `Read-File()` to send and receive data. In addition the driver supports timeouts and waiting for events like a serial line.
Special CAN related functions are available as I/O control commands via function `De-viceIoControl()`. In this document, we will just use the I/O control command name to refer to a call of `DeviceIoControl()` with this command value.
The following table lists all available functions for the CAN driver. They are discussed in detail in the next chapter.

Function	Description
`CreateFile()`	Opens the CAN interface for READ and/or WRITE access.
`CloseHandle()`	Closes the previously opened CAN interface.
`ReadFile()`	Reads the next event (as formatted text) from the queue.
`WriteFile()`	Sends a message (formatted as text) to the CAN bus.
`DeviceIoControl()`	Calls a special control function of the CAN driver.
`SetCommTimeouts()`	Changes Timeouts for read and write access.
`GetCommTimeouts()`	Returns current settings for read and write timeouts.
`SetCommMask()`	Sets the mask which events are recorded in the queue.
`GetCommMask()`	Returns current event mask.
`WaitCommEvent()`	Waits for an event.

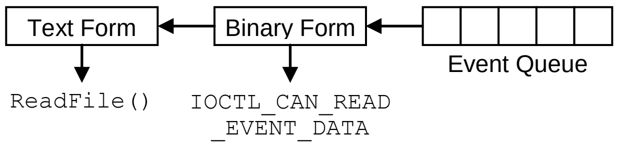
Table 8: Driver Interface Functions

7.2 Textual and Binary Message Representation

The driver has two different ways of sending data. When using function `WriteFile()`, the data to send must be prepared as a specially formatted text, for example by using `fprintf()` or similar functions. Sending data directly as a binary structure is possible with `IOCTL_CAN_WRITE_TRANSMIT_DATA`. Which way you choose depends on your preferences, both ways will send the same data frame over the CAN bus.

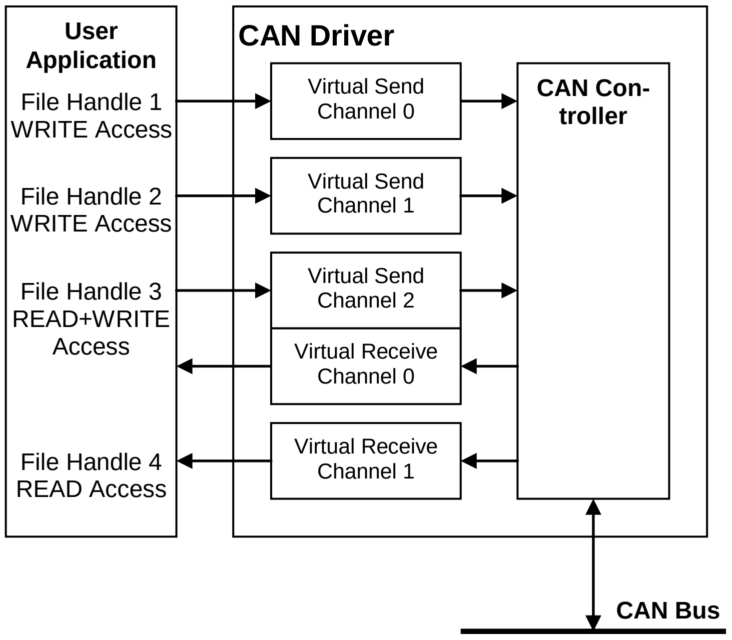


Every event happening on the CAN bus is reported to the driver and is stored in an event queue. Reading from the CAN driver means reading the next event from the event queue. There are also two different ways of reading this event data. Function `ReadFile()` returns a textual representation of the data and `IOCTL_CAN_READ_EVENT_DATA` returns a binary structure with the data. Again which way you choose is up to you. If on one hand the received message only needs to be output, then the textual representation may directly suit your needs. On the other hand, if different types of events should be distinguished, it is usually easier using the binary way as it avoids having to parse the textual data first.

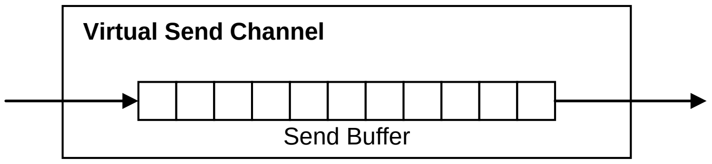


7.3 Virtual Send and Receive Channels

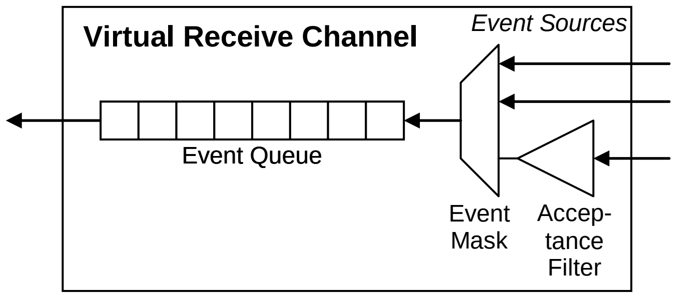
Sending and receiving through the CAN driver is completely independent from each other. You can open the CAN device several times and each file handle will behave as if it has an own direct channel to the CAN bus. We call this virtual send and virtual receive channels.


User Application
File Handle 1 WRITE Access
File Handle 2 WRITE Access
File Handle 3 READ+WRITE Access
File Handle 4 READ Access
CAN Driver
Virtual Send Channel 0
Virtual Send Channel 1
Virtual Send Channel 2
Virtual Receive Channel 0
Virtual Receive Channel 1
CAN Con-troller
CAN Bus

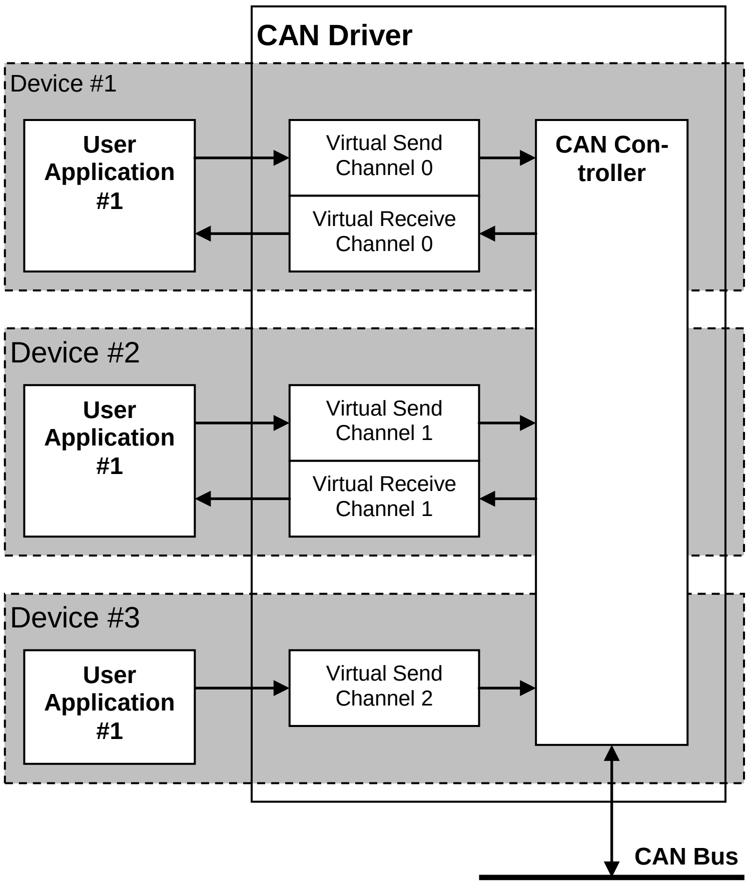
The number of available virtual channels depends on the CAN controller hardware. On the PicoCOM1, PicoCOM2 and PicoMOD3 you can open up to three virtual send channels and up to two virtual receive channels, because the CAN controller has three distinct transmit registers and two distinct receive registers.
Every time you open the CAN device with WRITE access (using `GENERIC_WRITE`), you get a new virtual send channel to the CAN bus with an own send buffer. The size of the send buffer can be configured in the registry. If the send buffer is full, the send function will block, unless a timeout is specified.



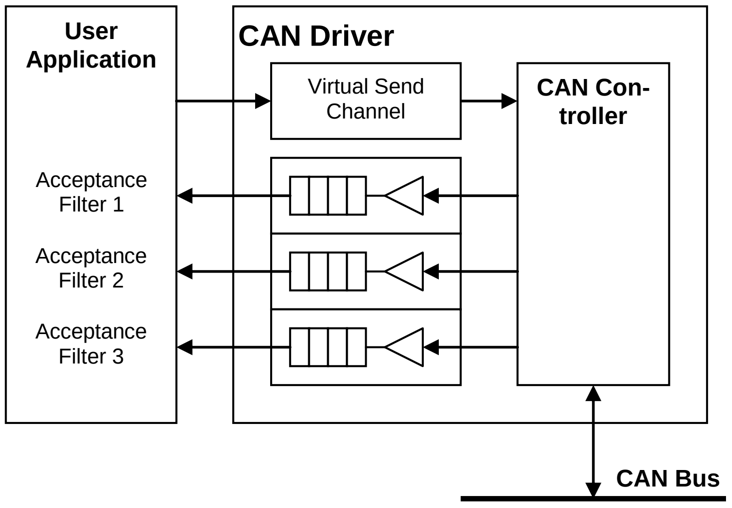
Every time you open the CAN device with READ access (using `GENERIC_READ`), you get a new virtual receive channel to the CAN bus with own event queue, acceptance filter and event mask. The size of the event queue can be configured in the registry.



The possibility to have an own acceptance filter for each virtual receive channel allows different applications to use the CAN bus as if they were separate CAN devices attached separately to the CAN bus.

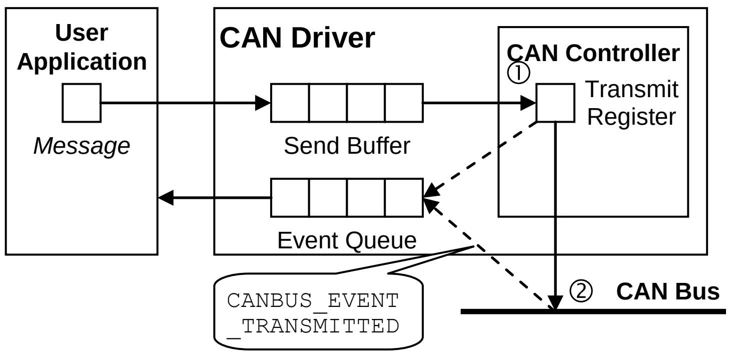


Or it allows one application to react to completely different acceptance filters that can not be represented as a single pair of acceptance code and mask.

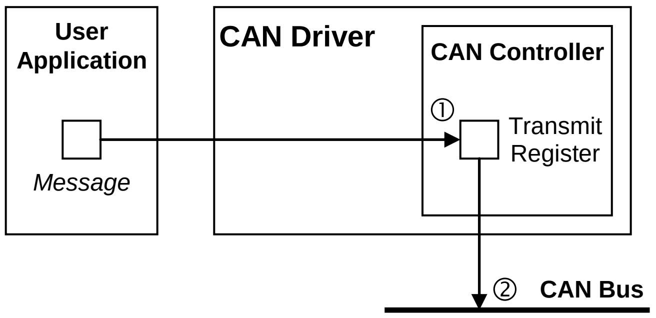


7.4 Send Buffer vs. Single Message Mode

The CAN driver can either run in Send Buffer Mode or in Single Message Mode (see registry entry `TxMode`).
In *Send Buffer Mode*, the sending function simply stores the message to send in the send buffer and then returns immediately. The rest is done by the background CAN service thread. It takes the message from the send buffer and puts it into the transmit register of the CAN controller.
The only way to tell if and when a message was actually sent is by activating `CANBUS_EVENT_TRANSMITTED` events.



When *Single Message Mode* is active, no send buffer is used. Instead the sending function itself puts the message into the transmit register of the CAN controller. If this register is still busy from before, it has to wait until the previous message is transmitted.



The point of time when a message is considered as transmitted can also be configured in `TxMode`. This can either be the time when the message is stored in the transmit register of the CAN controller ①, or the time when the message is completely transmitted on the CAN bus ②. At this configured point of time, event `CANBUS_EVENT_TRANSMITTED` is generated and the sending function in Single Message Mode returns.

7.5 Available Events

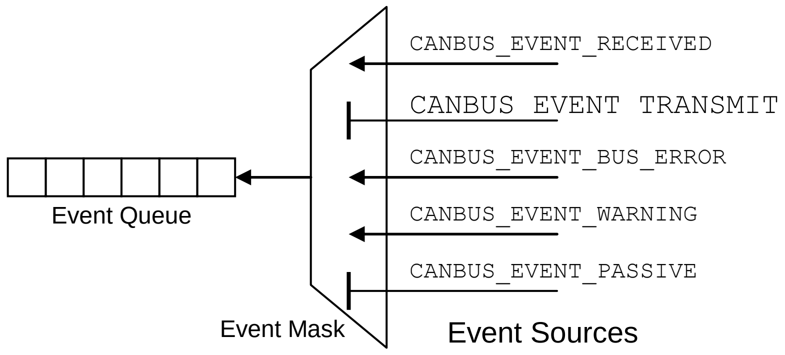
Here is a list of possible events generated by the CAN driver.

Event name	Description
`CANBUS_EVENT_RECEIVED`	A message was received and accepted
`CANBUS_EVENT_TRANSMITTED`	A message was successfully transmitted
`CANBUS_EVENT_ABORTED`	A transmit operation was aborted
`CANBUS_EVENT_ARBITRATION_LOST`	A message to be sent lost bus arbitration
`CANBUS_EVENT_DEVICE_CHANGED`	The CAN controller was set to a new configuration
`CANBUS_EVENT_ENTERING_STANDBY`	The CAN controller was switched to sleep mode
`CANBUS_EVENT_LEAVING_STANDBY`	The CAN controller was awaken from sleep mode
`CANBUS_EVENT_OVERRUN`	The receive register of the CAN controller could not take a message because the previous message was not yet fetched by the driver
`CANBUS_EVENT_WARNING`	There were several frame errors on the bus
`CANBUS_EVENT_PASSIVE`	Even more frame errors; the CAN controller switched to passive mode
`CANBUS_EVENT_BUS_ERROR`	Still more errors, the CAN controller went offline

Table 9: CAN Events

7.6 Event Mask

By setting an *event mask* with `SetCommMask()`, it is possible to tell the driver which types of events are interesting to the application and should actually be stored in the queue. All other events are discarded then. This may help reducing the number of events having to react to. The event mask is stored for each virtual receive channel separately.



As the event queue itself is influenced by the event mask, this impacts all available read functions.

Remark
When switching with `SetCommMask()` to a mask with fewer event types than before, and there were events waiting in the queue with these now missing types, these events are immediately deleted from the queue. They are irrevocably lost. Even switching back to the previous event mask will not bring them back again.

7.7 Counter for Lost Events

All events not masked by `SetCommMask()` are stored in the event queue. On a fast CAN bus with lots of traffic this may happen rather fast. Therefore the user application has to fetch the events from the event queue as fast as possible. If events arrive faster than they are taken from the queue, the queue will fill more and more until it is eventually completely full.
If still more events arrive when the queue is already full, these events are lost. But at least the driver counts these lost messages. When there is room again in the queue and the next event can be stored, the driver will save this count in a special *lost* field that is available in the event data structure. Therefore by looking at this field it is possible to tell how many events were lost since the last message that could be stored in the queue before.
In a normal scenario, the lost counters should be all zero. If an event shows a lost count of non-zero, this is an alarming sign that your application has reached its maximum capacity of handling CAN bus events.

7.8 Timeouts

When the send buffer is full, a sending function will block until there is again room in the send buffer. If the event queue is empty, a receiving function will block until an event arrives.
This behaviour can be influenced by setting timeouts with `SetCommTimeouts()`. Then the function will wait at most for the specified time and then returns, even if the function did not succeed. In this case the return value indicates an error state and `GetLastError()` will show `ERROR_TIMEOUT`.

7.9 Frame ID and Acceptance Filter Alignment

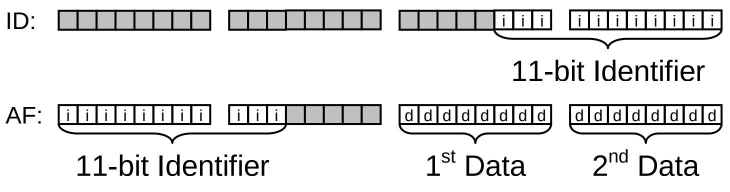
The CAN driver uses the current frame format and CAN bus mode to decide whether to send and accept Standard Frames or Extended Frames.

Mode	Format	Used Frame Size
`CANBUS_FORMAT_CAN_2_0_A`	(any)	Standard Frames
`CANBUS_FORMAT_CAN_2_0_B`	`CANBUS_TRANS_FMT_STD`	Standard Frames
	`CANBUS_TRANS_FMT_EXT`	Extended Frames
	`CANBUS_TRANS_FMT_DEFAULT`	Extended Frames

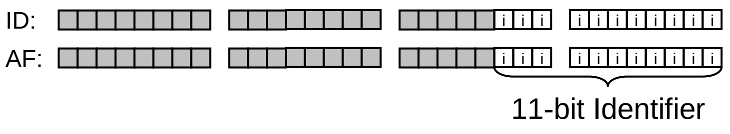
Table 10: ID and Filter Alignment

When switching the CAN bus mode and/or the frame format, the acceptance filters should be updated, too.
The alignment within the 32-bit values of the identifier (ID) and the acceptance filter values (AF) depends on the frame format and the registry setting `Align`.

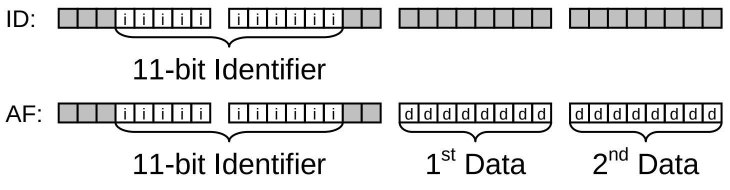
Accepting Standard Frames
In addition to the 11 bits of the Standard Frame ID, the driver allows to extend the acceptance filter up to the first two data bytes. The code and mask for these two bytes are given in bit 15..0. If you don't want to check these bytes, simply set the acceptance mask bits to "always accept".
`Align=0`: Mixed Alignment



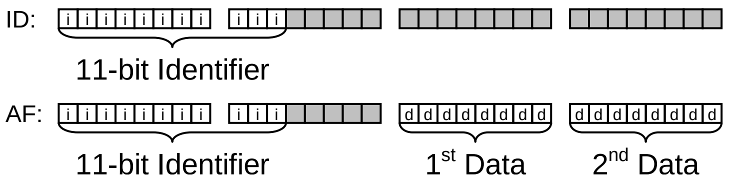
`Align=1`: Align at bit 10 (no data matching possible)



`Align=2`: Align at bit 28

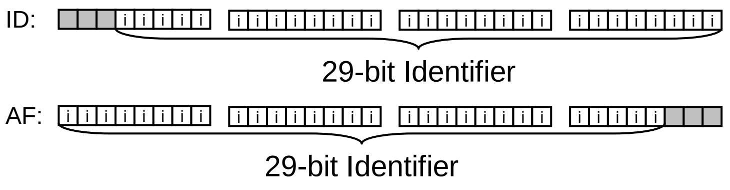


`Align=3`: Align at bit 31

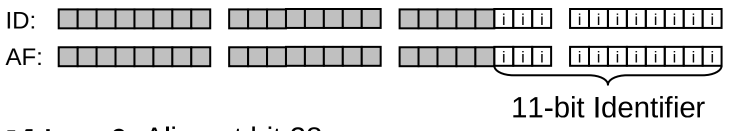


Accepting Extended Frames
Here the acceptance filter simply checks the 29 identifier bits.

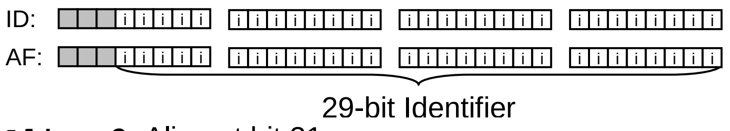
`Align=0`: Mixed Alignment



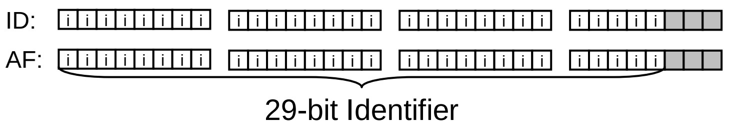
`Align=1`: Align at bit 10 (Remaining bits taken as zero)



`Align=2`: Align at bit 28



`Align=3`: Align at bit 31



7.10 New Features of Version 2.x

Version 2.x is a new and completely rewritten version of the CAN driver that was common on quite a few NetDCU boards of F&S before. We tried to keep the interface as similar as possible to the previous versions and only introduced new features that do not destroy compatibility.
The 1.x versions were mainly targeted to the SJA1000 CAN controller used on these previous boards. This was also reflected in some special function calls and commands that are not available on other controllers, and also the documentation often cited the SJA1000 controller directly.
The new version moves the interface to a more generic view of the controller, independent of any specific hardware. By the concept of virtual send and receive channels, the available separate hardware transmit and receive registers are mapped to separate file handles. The send buffers and event queues are made file handle specific, one buffer and/or queue for each open file handle. Sending and receiving are completely independent from each other now.
The `CANBUS_EVENT_ABORTED` is new in V2.x. Also some registry settings:

- `TxMode` to define the transmission configuration
- `SendBufferSize` to set the size of the send buffer,
- `AcceptanceCode` and `AcceptanceMask` to set a default acceptance filter when opening a virtual receive channel,
- `MaskActive` to define the bit logic of the acceptance mask, and
- `Align` to define the alignment of IDs and masks of messages and acceptance filters.

The defaults of these registry settings are chosen in a way that not setting them brings the same behaviour as in V1.x.
Opening the CAN device several times did not work properly (if at all) in previous versions. This is working now and is also thread-save, i.e. many different threads can read from and write to the same file handle without interfering with each other or causing other trouble within the driver. This required quite a lot of synchronisation work, especially when aborting transmissions or when closing file handles, as there always may be a couple of threads still waiting for service completion anywhere in the driver that need to be released before the data structures can be freed.

8 CAN Driver Reference

8.1 CreateFile()

Signature:

```
HANDLE CreateFile(
   LPCTSTR lpFileName, DWORD dwAccess,
   DWORD dwShareMode,
   LPSECURITY_ATTRIBUTES lpSecurity,
   DWORD dwCreation, DWORD dwFlags,
   HANDLE hTemplate
);
```

Parameters:

`lpFileName`	Device file name, usually “`CID1:`”
`dwAccess`	Device access (see below)
`dwShareMode`	File share mode (see below)
`lpSecurity`	Ignored, set to `NULL`
`dwCreation`	Set to `OPEN_EXISTING`
`dwFlags`	Set to `FILE_FLAG_WRITE_THROUGH`
`hTemplate`	Ignored, set to `0`

Device access `dwAccess`:

`0`	Device query mode
`GENERIC_READ`	Open a virtual receive channel only
`GENERIC_WRITE`	Open a virtual send channel only
`GENERIC_READ | GENERIC_WRITE`	Open virtual send & receive channel

File share mode `dwShareMode`:

`FILE_SHARE_READ`	Subsequent open operations succeed only if read access
`FILE_SHARE_WRITE`	Subsequent open operations succeed only if write access

Return:

`INVALID_HANDLE_VALUE`	Error, see `GetLastError()` for details
Otherwise	File handle

Description:

Opens the CAN device for the given access and returns a file handle. This file handle is required for all other functions using this CAN bus. The CAN controller is initialised with the registry values. It is *not* necessary to call `IOCTL_CAN_INIT` after opening the CAN port.
The driver will always use the next free virtual channel. Trying to open the device more often with a specific access mode than there are virtual channels available by the hardware, will fail with an error.
A virtual receive channel will use the default acceptance filter `AcceptanceCode` and `AcceptanceMask` from the registry and an event mask with all events enabled, unless new values are set with `IOCTL_CAN_WRITE_ACCEPTANCE_FILTER` and `SetCommMask()`.
If the file handle is not required anymore, you have to call function `CloseHandle()`.

8.2 CloseHandle()

Signature:
`BOOL CloseHandle(HANDLE hDevice);`

Parameters:

`hDevice`	Handle to device file

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Closes the device file that was opened with `CreateFile()`. This frees the virtual channels that were associated with the file handle. They can be re-used by issuing another call to `CreateFile()`.

8.3 WriteFile()

Signature:

```
BOOL WriteFile(
   HANDLE hDevice, LPVOID lpBuffer,
   DWORD dwCount,   LPDWORD lpdwWritten,
   LPOVERLAPPED lpOverlapped
);
```

Parameters:

`hDevice`	Handle to already open device file that must have WRITE access (opened with `GENERIC_WRITE`)
`lpBuffer`	Pointer to the data to send
`dwCount`	Number of bytes in `lpBuffer`
`lpdwWritten`	Pointer to a variable where the number of actually used bytes is stored.
`lpOverlapped`	Unused, set to `NULL`

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
This function sends a message to the CAN bus. It must get the data as a formatted ASCII text string and converts it to a binary CAN message before sending it. The transmission is configured by registry entry `TxMode`.
In *Single Message Mode*, `WriteFile()` blocks until the message is actually transmitted. In *Send Buffer Mode*, the message is only stored in the send buffer and `WriteFile()` returns immediately. The background CAN service routine then takes it from there and transmits it on the CAN bus as soon as possible. If the send buffer is full, `WriteFile()` blocks until there is room in the send buffer again for the message to be stored.
If configured in `TxMode` and if the file handle was opened with WRITE *and* READ access, the driver will generate an event `CANBUS_EVENT_TRANSMITTED` on the corresponding virtual receive channel, when the message was finally transmitted.
With `SetCommTimeouts()` it is possible to determine how long `WriteFile()` will wait at most when blocked. After this time, it will return with an error and `GetLastError()` will show `ERROR_TIMEOUT`.
You can use `IOCTL_CAN_WRITE_TRANSMIT_DATA` instead to transmit a message directly in binary form.

Message Format:
A message for `WriteFile()` has the following textual form:
`<id> <rtr> <dlc> <msg0> <msg1>... <eol>`
All values have to be given as hex numbers (without leading `0x` and without the angular brackets) and must be separated by non-hex characters, usually commas or white-space like `<TAB>` or blank characters.
The fields have the following meaning.

`<id>`	CAN identifier; the alignment must be as configured with registry entry `Align`
`<rtr>`	Remote Transmission Request: 0: CAN message with data 1: Request data from receiver
`<dlc>`	Number of data bytes (0..8)
`<msg0>`, `<msg1>`, ...	Up to 8 data bytes. Only the first `<dlc>` bytes of them are used if there are more bytes given
`<eol>`	Any combination of `\r`, `\n` and `\0` (i.e. Carriage Return `<CR>`, Line Feed `<LF>` and End-Of-String zero).

Please note that the driver uses plain ASCII characters here, *not* Unicode characters.
It is theoretically possible to combine more than one message in one call to `WriteFile()` or to split a message text into several calls to `WriteFile()`. However it is recommended to prepare exactly one message per call.

Example 1:
Send a message with ID `0x731`, <rtr>=`0`, and four data bytes `0x11`, `0x22`, `0x33` and `0x44`.

```
BYTE str[] = "731 0 4 11 22 33 44\n";
WriteFile(hCAN, str, strlen(str), &dwWritten,
          NULL);
```

Example 2:
This is a function that sends a generic message with 2 data bytes.

```
void SendCanMessage(HANDLE hCAN, DWORD dwID,
           BOOL bRTR, BYTE chData1, BYTE chData2)
{
      BYTE str[80];
      DWORD dwWritten
      sprintf(str, "%x %x %x %x %x %x\n", dwID,
             bRTR ? 1 : 0, 2, chData1, chData2);
      WriteFile(hCAN, str, strlen(str),
             &dwWritten, NULL);
}
```

8.4 ReadFile()

Signature:

```
BOOL ReadFile(
   HANDLE hDevice, LPVOID lpBuffer,
   DWORD dwCount,   LPDWORD lpdwRead,
   LPOVERLAPPED lpOverlapped
);
```

Parameters:

`hDevice`	Handle to already open device file that must have READ access (opened with `GENERIC_READ`)
`lpBuffer`	Pointer to a buffer where the received data will be stored
`dwCount`	Size of this buffer (in bytes); it should be large enough to hold a full event entry in text representation.
`lpdwRead`	Pointer to a variable where the number of actually received bytes is stored.
`lpOverlapped`	Unused, set to `NULL`

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
This function gets the next event from the event queue and returns it as a formatted ASCII text string.
If the event queue is empty, `ReadFile()` blocks until there is an event available. You can influence this behaviour by setting a timeout with `SetCommTimeouts()`.
You can use command `IOCTL_CAN_READ_EVENT_DATA` instead to receive an event directly in binary format.

Event Format:
An event consists of a string field telling the event type and a time field telling when the event has occurred. Some events have additional fields appended, for example the message that was received.
Please note that the driver uses plain ASCII characters, *not* Unicode characters.

- Event `CANBUS_EVENT_RECEIVED`:
  `received\t<time_high>:<time_low>\t<id>\t<rtr>`
  `\t<dlc>\r\n\t<msg0>\t<msg1>...\t<lost>\n`
- Event `CANBUS_EVENT_TRANSMITTED`:
  `transmitted\t<time_high>:<time_low>\t<id>\t<rtr>`
  `\t<dlc>\r\n\t<msg0>\t<msg1>...\n`
- Event `CANBUS_EVENT_ABORTED`:
  `aborted\t<time_high>:<time_low>\t<id>\t<rtr>`
  `\t<dlc>\r\n\t<msg0>\t<msg1>...\n`
- Event `CANBUS_EVENT_ARBITRATION_LOST`:
  `arbitration lost\t<time_high>:<time_low>\n`
- Event `CANBUS_EVENT_DEVICE_CHANGED`:
  `device changed\t<time_high>:<time_low>\n`
- Event `CANBUS_EVENT_ENTERING_STANDBY`:
  `entering standby\t<time_high>:<time_low>\n`
- Event `CANBUS_EVENT_LEAVING_STANDBY`:
  `leaving standby\t<time_high>:<time_low>\n`
- Event `CANBUS_EVENT_OVERRUN`:
  `overrun\t<time_high>:<time_low>\n`
- Event `CANBUS_EVENT_WARNING`:
  `warning\t<time_high>:<time_low>\n`
- Event `CANBUS_EVENT_PASSIVE`:
  `passive\t<time_high>:<time_low>\n`
- Event `CANBUS_EVENT_BUS_ERROR`:
  `bus error\t<time_high>:<time_low>\n`
- Any other event:
  `unknown event <ev>\t<time_high>:<time_low>\n`

Here the fields have the following meaning:

`<time_high>`	High-word of the time when the event occurred (32 bit as decimal value)
`<time_low>`	Low-word of this time (32 bit as decimal value)
`<id>`	CAN identifier of the message (32 bit value as 8 hex digits, aligned as configured by registry entry `Align`)
`<rtr>`	Remote transmission request (decimal value) 0: Normal CAN message 1: An answer is requested
`<dlc>`	Data length code (0..8 as decimal value)
`<msg0>`, `<msg1>`, ...	Up to eight message bytes (each 2 hex digits)
`<lost>`	Number of lost events since previous recorded event (32 bit as decimal value)

`<ev>`	Event number of the unknown event
`\t`	Tabulator character `<TAB>` (=0x09)
`\r`	Carriage return character `<CR>` (=0x0D)
`\n`	Line feed character `<LF>` (=0x0A)

Example 1:
The message with ID `0x731`, <rtr>=`0`, and four data bytes `0x11`, `0x22`, `0x33` and `0x44` was successfully transmitted at time `12345`.
```
transmitted  0:12345  731  0  4  11  22  33  44
```

Example 2:
A message with ID `0x112` and <rtr>=`1` with no data bytes was received at time `12568`. There were five messages lost before this message.
```
received  0:12568  112  1  0  5
```

Example 3:
The controller settings were changed at time `13210`.
```
device changed  0:13210
```

Please note that the larger blanks shown here in these examples are actually `<TAB>` characters `\t`, and the strings are closed by a linefeed character `\n`.

8.5 SetCommTimeouts()

Signature:
```
BOOL SetCommTimeouts(
   HANDLE hDevice,
   LPCOMMTIMEOUTS lpTimeouts
);
```

Parameters:

`hDevice`	Handle to already open device file
`lpTimeouts`	Pointer to a `COMMTIMEOUTS` structure with the timeout values to set

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
This function sets the timeout values used in all send and receive functions. Each open file handle can have its own set of timeout values.
A send function will block in Single Message Mode or when the send buffer is full. A receive function will block, when the event queue is empty. This means that these functions would not return until some activity on the CAN bus is finished.
By setting a timeout, these functions will return in any case after a specified maximum time, even if they did not succeed. The timeouts are set in a `COMMTIMEOUTS` structure with the following layout.
```
typedef struct _COMMTIMEOUTS
{
   DWORD ReadIntervalTimeout;
   DWORD ReadTotalTimeoutMultiplier;
   DWORD ReadTotalTimeoutConstant;
   DWORD WriteTotalTimeoutMultiplier;
   DWORD WriteTotalTimeoutConstant;
} COMMTIMEOUTS, *LPCOMMTIMEOUTS;
```

This structure, usually used with serial lines, defines the maximum times allowed between two bytes (interval timeouts) and for the whole serial message (total timeouts).
With this CAN driver, messages are received as events. This makes it impossible to define read timeouts that depend on the message length. An event is either available or not. Therefore the two entries `ReadIntervalTimeout` and `ReadTotalTimeoutMultiplier` are ignored by the driver and the timeout is set by `ReadTotalTimeoutConstant` only.
Entry `WriteTotalTimeoutMultiplier` is combined with the data length code DLC, i.e. the number of data bytes in the final binary message, *not* the length of the textual representation given by `dwCount` in `WriteFile()`, as this would not make sense. The final timeout used is

`WriteTotalTimeoutMultiplier` * DLC
`+ WriteTotalTimeoutConstant`.

All timeout values are given in milliseconds. The value `INFINITE` has the special meaning of “wait infinitely”. Therefore to avoid timeouts, set `ReadTotalTimeoutConstant` and/or `WriteTotalTimeoutConstant` to `INFINITE`.
A timeout value of zero means return immediately. This is different to serial lines where it means “don’t use timeouts”. As this would be the same as “wait infinitely”, zero is much better used for completely non-blocking calls.

You can use function `GetCommTimeouts()` to determine the currently active timeout settings.

Example:
Change the read timeout to 300ms, keep all other timeouts.

```
COMMTIMEOUTS cTimeouts;
HANDLE hCAN;
hCAN = CreateFile(...);
GetCommTimeouts(hCAN, &cTimeouts);
cTimeouts.ReadTotalTimeoutConstant = 300;
SetCommTimeouts(hCAN, &cTimeouts);
```

8.6 GetCommTimeouts()

Signature:
```
BOOL GetCommTimeouts(
   HANDLE hDevice,
   LPCOMMTIMEOUTS lpTimeouts
);
```

Parameters:

`hDevice`	Handle to already open device file
`lpTimeouts`	Pointer to a `COMMTIMEOUTS` structure where the current timeout values will be stored

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
This function returns the timeout values that are currently active in the driver. For a description of the `COMMTIMEOUTS` structure and an example see function `SetCommTimeouts()` on page 39. Entries `ReadIntervalTimeout` and `ReadTotalTimeoutMultiplier` are not used by the driver and can safely be ignored.

8.7 SetCommMask()

Signature:

```
BOOL SetCommMask(
   HANDLE hDevice,
   DWORD dwEventMask,
);
```

Parameters:

`hDevice`	Handle to already open device file
`dwEventMask`	Specifies the events to enable (see table below)

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
This function tells the driver which types of events should be recorded in the event queue. All other events are ignored from then on. See page 25 for a list of possible event types.
If the new mask that is set with `SetCommMask()` contains fewer or other event types than before and there were already events waiting in the queue with a type now disabled, these events are immediately removed from the queue. They are irrevocably gone and therefore counted as “lost” in the next remaining entry.
Each virtual receive channel, i.e. file handle with READ access, can have its own event mask. You can use function `GetCommMask()` to determine the currently active event mask.

Example:
Lets assume someone has misconfigured the CAN driver to use a far too small event queue with only eight entries. In fact this event queue is so small that it is nearly always full and events don’t fit in rather often. Therefore those events that are lucky to be recorded have significant lost counts.
We start our observation with the following content of the event queue.

#	CANBUS_EVENT_	Lost	Meaning
1	RECEIVED	2	Two events were lost before this message was received
2	RECEIVED	5	Five events were lost between #1 and #2
3	DEVICE_CHANGED	4	Device was reconfigured, four other events were lost since #2
4	DEVICE_CHANGED	3	Again a reconfiguration was recorded after three events were lost
5	TRANSMITTED	1	A message was successfully transmitted, but one lost event
6	DEVICE_CHANGED	2	Two lost messages since #5
7	RECEIVED	0	This is the event immediately following #6
8	TRANSMITTED	0	And this directly followed #7

Now the following commands are executed and remove the event type `CANBUS_EVENT_DEVICE_CHANGED` from the active event mask.

```
DWORD dwEventMask;
GetCommMask(hCAN, &dwEventMask);
dwEventMask &= ~CANBUS_EVENT_DEVICE_CHANGED;
SetCommMask(hCAN, dwEventMask);
```

This will immediately delete entries #3, #4 and #6 from the queue. Their lost counts (plus the number of the deleted events themselves) are added to the remaining event entries.
Let's look at the new content of the queue immediately after `SetCommMask()`. Note how the lost counts have changed.

#	`CANBUS_EVENT_`	Lost	Meaning
1	`RECEIVED`	2	As before
2	`RECEIVED`	5	As before
3	`TRANSMITTED`	10	This is the previous entry #5 that had a lost count of 1. Now there are two additional events lost (#3 and #4), and these events had an own lost count of 4 and 3. This sums up to the new lost count of 1+2+4+3=10.
4	`RECEIVED`	3	This is previous entry #7 which had a lost count of 0. Now as previous #6 is deleted, we have to add its lost count of 2 here, plus the now missing #6 itself. This sums up to 0+1+2=3.
5	`TRANSMITTED`	0	This is previous entry #8, no change
6	-	0	This entry is now empty
7	-	0	This entry is now empty
8	-	0	This entry is now empty

Remark:
This is a constructed example. Nobody would actually use such an insufficient and unreliable configuration in real life. Usually the lost counts are all zero.
If an event shows a lost count of non-zero that is not the result of the removal of entries by `SetCommMask()` as outlined above, this is an alarming sign that your application has already reached its maximum capacity of handling CAN bus events.

8.8 GetCommMask()

Signature:
```
BOOL GetCommMask(
   HANDLE hDevice,
   LPDWORD lpdwEventMask,
);
```

Parameters:

`hDevice`	Handle to already open device file
`lpdwEventMask`	Pointer to a `DWORD` where the current event mask will be stored

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Return the event mask that is currently active in the driver.
Use function `SetCommMask()` to set a new event mask. See page 42 for a detailed description and an example.

8.9 WaitCommEvent()

Signature:
```
BOOL WaitCommEvent(
   HANDLE hDevice,
   LPDWORD lpdwEvent,
   LPOVERLAPPED lpOverlapped
);
```

Parameters:

`hDevice`	Handle to already open device file
`lpdwEvent`	Pointer to a `DWORD` where the event type will be stored
`lpOverlapped`	Unused, set to `NULL`

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
This function waits until an event is available in the event queue. It keeps the event in the queue but returns the event type in the `DWORD` where `lpdwEvent` points to. If this value is zero, this indicates that the function was aborted, for example because the device was closed. Check `GetLastError()` in this case.
If there are already events waiting in the event queue when `WaitCommEvent()` is called, it simply peeks into the type of the first queue entry and returns immediately.
You can use `SetCommMask()` to define the set of event types to react on and `Get-CommMask()` to get the currently active set of event types.
Please note that `WaitCommEvent()` does not use the timeouts set with `Set-CommTimeouts()`. It will wait indefinitely if no event arrives.

Example:
Wait until the CAN controller is woken up from sleep mode by some external bus activity.

```
DWORD dwEvent;
HANDLE hCAN;
BYTE chCmd = CANBUS_CMD_ENTER_STANDBY;
hCAN = CreateFile(...);
...
/* Set controller into sleep mode */
DeviceIoControl(hCAN, IOCTL_CAN_SET_COMMAND,
      &chCMD, sizeof(chCMD), NULL, 0, NULL, NULL);
/* Wait until controller wakes up again */
SetCommMask(hCAN, CANBUS_EVENT_LEAVING_STANDBY);
WaitCommEvent(hCAN, &dwEvent, NULL);
```

8.10 DeviceIoControl()

Signature:

```
int DeviceIoControl(
   HANDLE hDevice, DWORD dwIoControlCode,
   LPVOID lpInBuffer, DWORD nInBufferSize,
   LPVOID lpOutBuffer, DWORD nOutBufferSize,
   LPDWORD lpReturned, LPOVERLAPPED lpOverlapped
);
```

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	Control code specifying the device specific command to execute
`lpInBuffer`	Pointer to the data going into the function (`IN` data)
`nInBufferSize`	Size of the `IN` data (in bytes)
`lpOutBuffer`	Pointer to a buffer where data coming out of the function can be stored (`OUT` data)
`nOutBufferSize`	Number of bytes available for the `OUT` data
`lpReturned`	Number of bytes actually written to the `OUT` data buffer
`lpOverlapped`	Unused, set to `NULL`

Description:

Executes a device specific function. The type of function is given by a control code in `dwIo-ControlCode`. Each function has a specific set of parameters. Usually there is some data going into the function (`IN` data) and some data is returned out of the function (`OUT` data). The following table lists all control codes recognised by the CAN driver. These codes as well as all required data structures are defined in the header file `canbusio.h` (see page 74) and will be explained in detail in the following subsections.

Control Code	Function
`IOCTL_CAN_WRITE _ACCEPTANCE_FILTER`	Set new acceptance filter for the virtual re-ceive channel
`IOCTL_CAN_READ _ACCEPTANCE_FILTER`	Get the current acceptance filter of the re-ceive channel
`IOCTL_CAN_SET_BAUDRATE`	Set a new CAN bus baud rate
`IOCTL_CAN_GET_BAUDRATE`	Get the current baud rate
`IOCTL_CAN_SET_BAUDRATE _BY_CONSTANT`	Select a predefined baud rate from the baud rate table
`IOCTL_CAN_GET_BAUDRATE _BY_CONSTANT`	Get the index into the baud rate table for the current rate
`IOCTL_CAN_SET_CAN_MODE`	Set a new CAN bus mode
`IOCTL_CAN_SET_DEFAULT _FRAME_FORMAT`	Set a new CAN bus frame format
`IOCTL_CAN_WRITE_TRANSMIT_DATA`	Transmit a message in binary form

`IOCTL_CAN_READ_EVENT_DATA`	Read the next event in binary form
`IOCTL_CAN_READ_TIME`	Get the current CAN time
`IOCTL_CAN_READ_PROPERTIES`	Read the capabilities of the CAN controller and driver
`IOCTL_CAN_SET_COMMAND`	Execute a special CAN controller command
`IOCTL_CAN_READ_REGISTER`	Read a register of the CAN controller
`IOCTL_CAN_WRITE_REGISTER`	Write a value to a register of the CAN con-troller
`IOCTL_CAN_READ_REGISTER_RM`	Read a configuration register of the CAN controller
`IOCTL_CAN_WRITE_REGISTER_RM`	Write a configuration register of the CAN controller
`IOCTL_CAN_INIT`	Initialise the CAN controller
`IOCTL_CAN_TEST_DEVICE`	Check if the CAN controller is in Configura-tion Mode

Table 11: Function DeviceIiControl – Control Codes

8.11 IOCTL_CAN_WRITE_ACCEPTANCE_FILTER

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_WRITE _ACCEPTANCE_FILTER`
`lpInBuffer`	Pointer to structure `CAN_ACCEPTANCE_FILTER` with the new acceptance filter information
`nInBufferSize`	`sizeof(CAN_ACCEPTANCE _FILTER)`
`lpOutBuffer`	Unused, set to `NULL`
`nOutBufferSize`	Unused, set to `0`
`lpReturned`	Unused, set to `NULL`
`lpOverlapped`	Unused, set to `NULL`

Return:

0	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Set a new acceptance filter for the virtual receive channel. The acceptance filter determines which incoming messages are accepted and which are ignored, depending on the incoming message ID. Every virtual receive channel (i.e. open file handle with READ access) can have its own acceptance filter.
See page 7 for a detailed description of acceptance filters.
`CAN_ACCEPTANCE_FILTER` is a structure defined in header file `canbusio.h` with the following form:

```
typedef struct tagCAN_ACCEPTANCE_FILTER
{
     CAN_DWORD code;
     CAN_DWORD mask;
} CAN_ACCEPTANCE_FILTER, *PCAN_ACCEPTANCE_FILTER;
```

The `code` defines the value that every incoming message ID is compared to and the `mask` decides which bits of the incoming ID must match the code exactly and which bits are always accepted.
The format of the filter data is configured with registry settings `Align` and `MaskActive`. If no acceptance filter is set after opening the device, the default filter as given by registry entries `AcceptanceCode` and `AcceptanceMask` is used.

Example 1:

Set an acceptance filter for standard frames to accept IDs 0x120 to 0x127. We assume Align=1 and MaskActive=0 here.

```
HANDLE hCAN;
CAN_ACCEPTANCE_FILTER cFilter;
hCAN = CreateFile(..., GENERIC_READ, ...);
cFilter.code = 0x120;
cFilter.mask = 0x007;
DeviceIoControl(hCAN,
      IOCTL_CAN_WRITE_ACCEPTANCE_FILTER,
      &cFilter, sizeof(CAN_ACCEPTANCE_FILTER),
      NULL, 0, NULL, NULL);
```

To get the same setting with `Align=0`, you would have to use these values:

```
cFilter.code = 0x24000000;
cFilter.mask = 0x00E00000;
```

And with `Align=0` and `MaskActive=1` these:

```
cFilter.code = 0x24000000;
cFilter.mask = 0xFF1FFFF8;
```

Example 2:

Set an acceptance filter for extended frames to accept IDs `0x00234500` to `0x002345FF`, `0x01234500` to `0x012345FF`, `0x0234500` to `0x02345FF` and `0x0334500` to `0x03345FF`. We assume `MaskActive=0` now and `Align=2`, which is the best format for 29-bit IDs.

```
HANDLE hCAN;
CAN_ACCEPTANCE_FILTER cFilter;
hCAN = CreateFile(..., GENERIC_READ, ...);
cFilter.code = 0x00234500;
cFilter.mask = 0x030000FF;
DeviceIoControl(hCAN,
      IOCTL_CAN_WRITE_ACCEPTANCE_FILTER,
      &cFilter, sizeof(CAN_ACCEPTANCE_FILTER),
      NULL, 0, NULL, NULL);
```

8.12 IOCTL_CAN_READ_ACCEPTANCE_FILTER

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_READ _ACCEPTANCE_FILTER`
`lpInBuffer`	Unused, set to `NULL`
`nInBufferSize`	Unused, set to `0`
`lpOutBuffer`	Pointer to structure `CAN_ACCEPTANCE_FILTER` where the current acceptance filter information will be stored
`nOutBufferSize`	`sizeof(CAN_ACCEPTANCE _FILTER)`
`lpReturned`	If not `NULL`, the referenced `DWORD` will be set to the number of actually returned bytes in `lpOutBuffer`
`lpOverlapped`	Unused, set to `NULL`

Return:

0	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Read the currently active acceptance filter settings for the virtual receive channel. See page 48 for a description of structure `CAN_ACCEPTANCE_MASK`.

Example:

```
HANDLE hCAN;
CAN_ACCEPTANCE_FILTER cFilter;
DWORD dwCount;
hCAN = CreateFile(..., GENERIC_READ, ...);
DeviceIoControl(hCAN,
      IOCTL_CAN_READ_ACCEPTANCE_FILTER, NULL, 0,
      &cFilter, sizeof(CAN_ACCEPTANCE_FILTER),
      &dwCount, NULL);
/* cFilter contains acceptance filter now */
```

8.13 IOCTL_CAN_SET_BAUDRATE

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_SET_BAUDRATE`
`lpInBuffer`	Pointer to `DWORD` with the new rate
`nInBufferSize`	`sizeof(DWORD)`
`lpOutBuffer`	Unused, set to `NULL`
`nOutBufferSize`	Unused, set to `0`
`lpReturned`	Unused, set to `NULL`
`lpOverlapped`	Unused, set to `NULL`

Return:

0	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Set a new baud rate on the CAN bus. The allowed range can be determined with `IOCTL_CAN_READ_PROPERTIES`.
This is a global setting and will influence all virtual channels that are currently open. Therefore all virtual receive channels will get event `CANBUS_EVENT_DEVICE_CHANGED`.

8.14 IOCTL_CAN_GET_BAUDRATE

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_GET_BAUDRATE`
`lpInBuffer`	Unused, set to `NULL`
`nInBufferSize`	Unused, set to `0`
`lpOutBuffer`	Pointer to `DWORD` where the current baud rate will be stored
`nOutBufferSize`	`sizeof(DWORD)`
`lpReturned`	If not `NULL`, the referenced `DWORD` will be set to the number of actually returned bytes in `lpOutBuffer`
`lpOverlapped`	Unused, set to `NULL`

Return:

0	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Get the baud rate that is currently active.

8.15 IOCTL_CAN_SET_BAUDRATE_BY_CONSTANT

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_SET_BAUDRATE _BY_CONSTANT`
`lpInBuffer`	Pointer to `DWORD` with the baud rate table index of the new rate
`nInBufferSize`	`sizeof(DWORD)`
`lpOutBuffer`	Unused, set to `NULL`
`nOutBufferSize`	Unused, set to `0`
`lpReturned`	Unused, set to `NULL`
`lpOverlapped`	Unused, set to `NULL`

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Set a new baud rate from the table of predefined baud rates. The table of predefined baud rates can be determined with `IOCTL_CAN_READ_PROPERTIES`.
This is a global setting and will influence all virtual channels that are currently open. Therefore all virtual receive channels will get event `CANBUS_EVENT_DEVICE_CHANGED`.

8.16 IOCTL_CAN_GET_BAUDRATE_BY_CONSTANT

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_GET_BAUDRATE _BY_CONSTANT`
`lpInBuffer`	Unused, set to `NULL`
`nInBufferSize`	Unused, set to `0`
`lpOutBuffer`	Pointer to `DWORD` where the current baud rate table index will be stored
`nOutBufferSize`	`sizeof(DWORD)`
`lpReturned`	If not `NULL`, the referenced `DWORD` will be set to the number of actually returned bytes in `lpOutBuffer`
`lpOverlapped`	Unused, set to `NULL`

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Get the index of the currently active baud rate with respect to the table of predefined baud rates. If there is no exact match in the table, e.g. when set with `IOCTL_CAN_SET_BAUDRATE` and not `IOCTL_CAN_SET_BAUDRATE_BY_CONSTANT`, value `-1` (=`0xFFFFFFFF`) is returned.

8.17 IOCTL_CAN_SET_CAN_MODE

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_SET_CAN_MODE`
`lpInBuffer`	Pointer to `DWORD` with the new CAN bus operation mode (see below)
`nInBufferSize`	`sizeof(DWORD)`
`lpOutBuffer`	Unused, set to `NULL`
`nOutBufferSize`	Unused, set to `0`
`lpReturned`	Unused, set to `NULL`
`lpOverlapped`	Unused, set to `NULL`

CAN Bus Mode in `lpInBuffer`:

`CANBUS_FORMAT_CAN_2_0_A`	Use CAN2.0A mode
`CANBUS_FORMAT_CAN_2_0_B`	Use CAN2.0B mode

Return:

0	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Set new CAN bus mode. See also registry entry `CanMode2B`.
This is a global setting and will influence all virtual channels that are currently open. Therefore all virtual receive channels will get event `CANBUS_EVENT_DEVICE_CHANGED`.

8.18 IOCTL_CAN_SET_DEFAULT_FRAME_FORMAT

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_SET_DEFAULT _FRAME_FORMAT`
`lpInBuffer`	Pointer to `DWORD` with the format
`nInBufferSize`	`sizeof(DWORD)`
`lpOutBuffer`	Unused, set to `NULL`
`nOutBufferSize`	Unused, set to `0`
`lpReturned`	Unused, set to `NULL`
`lpOverlapped`	Unused, set to `NULL`

New Frame Format in `lpInBuffer`:

`CANBUS_TRANS_FMT_DEFAULT`	Use frame format as given by the current CAN bus operation mode: CAN2.0A: Standard Frames CAN2.0B: Extended Frames
`CANBUS_TRANS_FMT_STD`	Use Standard Frames
`CANBUS_TRANS_FMT_EXT`	Use Extended Frames

Return:

0	Error, see `GetLastError()` for details
`!=0`	Success

Description:

Set a new CAN bus frame format. This format is used for all send and receive functions. If a new frame format is set, all acceptance filters must be reconfigured, too.

This is a global setting and will influence all virtual channels that are currently open. Therefore all virtual receive channels will get event `CANBUS_EVENT_DEVICE_CHANGED`.

8.19 IOCTL_CAN_WRITE_TRANSMIT_DATA

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_WRITE _TRANSMIT_DATA`
`lpInBuffer`	Pointer to structure `CAN_TRANSMIT_DATA` with the message to send
`nInBufferSize`	`sizeof(CAN_TRANSMIT_DATA)`
`lpOutBuffer`	Unused, set to `NULL`
`nOutBufferSize`	Unused, set to `0`
`lpReturned`	Unused, set to `NULL`
`lpOverlapped`	Unused, set to `NULL`

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
This function sends a message given in binary form to the CAN bus. The transmission is configured by registry entry `TxMode`.
In *Single Message Mode*, the function blocks until the message is actually transmitted. In *Send Buffer Mode*, the message is only stored in the send buffer and the function immediately returns. The background CAN service routine then takes it from there and transmits it on the CAN bus as soon as possible. If the send buffer is full, this function blocks until there is room in the send buffer again for the message to be stored.
If configured in `TxMode` and if the file handle was opened with WRITE *and* READ access, the driver will generate an event `CANBUS_EVENT_TRANSMITTED` on the corresponding virtual receive channel, when the message was finally transmitted.
With `SetCommTimeouts()` it is possible to determine how long `IOCTL_CAN_WRITE_TRANSMIT_DATA` will wait at most when blocked. After this time, it will return with an error and `GetLastError()` will show `ERROR_TIMEOUT`.

`CAN_TRANSMIT_DATA` is a structure defined in `canbusio.h` with the following form:

```
typedef struct tagCAN_TRANSMIT_DATA
{
      CAN_UINT fmt;
      CAN_DWORD identifier;
      CAN_BYTE rtr;
      CAN_BYTE dlc;
      CAN_BYTE msg[8];
} CAN_TRANSMIT_DATA, *PCAN_TRANSMIT_DATA;
```

The fields have the following meaning.

`fmt`	The frame format to use (see below)
`identifier`	CAN identifier; the alignment must be as configured with registry entry `Align`
`rtr`	Remote Transmission Request: 0: CAN message with data 1: Request data from receiver
`dlc`	Number of data bytes (0..8)
`msg[]`	Up to 8 data bytes. Only the first `dlc` bytes of them are used if there are more bytes given

Field `fmt` can have one of the following values.

`fmt`	Meaning
`CANBUS_TRANS_FMT_DEFAULT`	Use the default frame format like any other read or write function.
`CANBUS_TRANS_FMT_STD`	Use a Standard Frame (*)
`CANBUS_TRANS_FMT_EXT`	Use an Extended Frame (*)

The values marked with (*) overrule the default format as set by `IOCTL_CAN_SET_DEFAULT_FRAME_FORMAT` or registry entry `Format`. This allows a more flexible way of sending frames of different size in CAN2.0B networks.

Example:
Send a Standard Frame with ID `0x123` and two data bytes `0x11` and `0x22`. After that, send an Extended Frame requesting data from ID `0x12345678`. We assume `Align=0` and CAN2.0B mode here.

```
HANDLE hCAN;
CAN_TRANSMIT_DATA cData;
hCAN = CreateFile(..., GENERIC_WRITE, ...);
...
cData.fmt = CANBUS_TRANS_FMT_STD;
cData.identifier = 0x123;
cData.rtr = 0;
cData.dlc = 2;
cData.msg[0] = 0x11;
cData.msg[1] = 0x22;
DeviceIoControl(hCAN,
      IOCTL_CAN_WRITE_TRANSMIT_DATA,
      &cData, sizeof(CAN_TRANSMIT_DATA),
      NULL, 0, NULL, NULL);
cData.fmt = CANBUS_TRANS_FMT_EXT;
cData.identifier = 0x12345678;
cData.rtr = 1;
cData.dlc = 0;
DeviceIoControl(hCAN,
      IOCTL_CAN_WRITE_TRANSMIT_DATA,
      &cData, sizeof(CAN_TRANSMIT_DATA),
      NULL, 0, NULL, NULL);
```

8.20 IOCTL_CAN_READ_EVENT_DATA

Parameters:

hDevice	Handle to already open device file
dwIoControlCode	IOCTL_CAN_READ_EVENT_DATA
lpInBuffer	Unused, set to NULL
nInBufferSize	Unused, set to 0
lpOutBuffer	Pointer to structure CAN_EVENT where the event data will be stored
nOutBufferSize	sizeof(CAN_EVENT)
lpReturned	If not NULL, the referenced DWORD will be set to the number of actually returned bytes in lpOutBuffer
lpOverlapped	Unused, set to NULL

Return:

0	Error, see GetLastError() for details
!=0	Success

Description:
This function gets the next event from the event queue and returns it in binary representation in a CAN_EVENT structure.
If the event queue is empty, the function blocks until there is an event available. With SetCommTimeouts() it is possible to determine how long IOCTL_CAN_READ_EVENT_DATA will wait at most when blocked. After this time, it will return with an error and GetLastError() will show ERROR_TIMEOUT.
CAN_EVENT is a structure defined in canbusio.h with the following form:

```
typedef struct tagCAN_EVENT
{
      CAN_DWORD event;
      CAN_TIME time;
      CAN_DWORD lost;
      CAN_TRANSMIT_DATA data;
      CAN_UINT arbitration;
} CAN_EVENT, *PCAN_EVENT;
```

The fields have the following meaning.

event	The type of the event (see page 25)
time	The CAN bus time when the event occurred; see page 62 for a description of CAN_TIME
lost	The number of lost events since the last recorded event, i.e. events that could not be recorded because the event queue was full
data	The message data. See page 57 for a description of structure CAN_TRANSMIT_DATA. This field is only valid if event is one of CANBUS_EVENT_RECEIVED, CANBUS_EVENT_TRANSMITTED or CANBUS_EVENT_ABORTED.
arbitration	Unused, ignore

You can use ReadFile() instead to receive an event in text format.

Example:
Read the next event and handle it. If a new message was received, look at the ID to decide further actions.

```
HANDLE hCAN;
CAN_EVENT cEvent;
hCAN = CreateFile(..., GENERIC_READ, ...);
...
DeviceIoControl(hCAN, IOCTL_CAN_READ_EVENT_DATA, NULL, 0,
      &cEvent, sizeof(CAN_EVENT), NULL, NULL);
switch (cEvent.event)
{
      case CANBUS_EVENT_TRANSMITTED:
            ...
            break;
      case CANBUS_EVENT_ABORTED:
            ...
            break;
      case CANBUS_EVENT_RECEIVED:
            switch (cEvent.data.identifier)
            {
                  ...
            }
            break;
      default:
            /* Ignore */
            break;
}
```

8.21 IOCTL_CAN_READ_TIME

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_READ_TIME`
`lpInBuffer`	Unused, set to `NULL`
`nInBufferSize`	Unused, set to `0`
`lpOutBuffer`	Pointer to a `CAN_TIME` structure where the current time will be stored
`nOutBufferSize`	`sizeof(CAN_TIME)`
`lpReturned`	If not `NULL`, the referenced `DWORD` will be set to the number of actually returned bytes in `lpOutBuffer`
`lpOverlapped`	Unused, set to `NULL`

Return:

0	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Get the current CAN bus time. The CAN bus time is a 64 bit value used to tell when an event occurred. As the time continuously goes on, every call to `IOCTL_CAN_READ_TIME` will return a different value.

The `CAN_TIME` structure is defined in `canbusio.h` and has the following form:

```
typedef struct tagCAN_TIME
{
      CAN_DWORD low;
      CAN_DWORD high;
} CAN_TIME, *PCAN_TIME;
```

The fields have the following meaning.

`low`	The lower 32 bits of the 64 bit time value
`high`	The higher 32 bits of the 64 bit timer value

Remark:
The current CAN driver implementation uses the Windows Tick Counter as CAN bus time. This is a 32 bit counter that starts when the power to the device is switched on and increments every millisecond. It is used as `low` field. The `high` field is always zero.
Please note that in this constellation the Windows Tick Counter and therefore the CAN bus time will wrap around to zero after 2^{32} ms = ~49.7 days of non-stop operation. It will *not* increment the `high` field of the CAN bus time then!

8.22 IOCTL_CAN_READ_PROPERTIES

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_READ_PROPERTIES`
`lpInBuffer`	Unused, set to `NULL`
`nInBufferSize`	Unused, set to `0`
`lpOutBuffer`	Pointer to a `CAN_PROPERTIES` structure where the CAN controller properties will be stored
`nOutBufferSize`	`sizeof(CAN_PROPERTIES)`
`lpReturned`	If not `NULL`, the referenced `DWORD` will be set to the number of actually returned bytes in `lpOutBuffer`
`lpOverlapped`	Unused, set to `NULL`

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
This CAN driver may be used on different boards with different hardware CAN controllers. By calling this function, the properties of the driver in connection with this specific CAN controller can be determined, e.g. name and version, range of baud rates, available commands, etc. Please note that this data is mode-specific. For example in CAN2.0B mode the controller may support more commands than in CAN2.0A mode.
The information is returned in a `CAN_PROPERTIES` structure which is defined in `canbusio.h` and has the following form:

```
typedef struct tagCAN_PROPERTIES
{
      CAN_DWORD version;
      TCHAR device_name[100];
      CAN_DWORD min;
      CAN_DWORD max;
      CAN_INT nCommands;
      CAN_INT commands[10];
      CAN_INT nBaudrates;
      CAN_DWORD baudrates[50];
      CAN_DWORD chipset_flags;
      CAN_INT nRegisters;
}
```

Here the fields have the following meaning.

`version`	Version of the driver. The version is divided in a major and a minor number with each one byte. For example in V2.1, 2 is the major and 1 is the minor version. `version = (major << 8) | minor`
`device_name`	Name and version of the driver and the CAN controller
`min`	Minimum baud rate that is possible
`max`	Maximum baud rate that is possible
`nCommands`	Number of valid entries in `commands[]`
`commands[]`	Table of available controller commands for `IOCTL_CAN_SET_COMMAND`. Each entry of this array holds one available command. See page 65 for a list of possible commands. Only `nCommands` entries are valid, ignore all remaining entries.
`nBaudrates`	Number of valid entries in baudrates[]
`baudrates[]`	Table of predefined baud rates. Each entry holds one possible baud rate that is guaranteed to work properly. Only `nBaudrates` entries are valid, ignore the remaining entries.
`chipset_flags`	A combination of flags showing the CAN bus capabilities (see below)
`nRegisters`	Number of registers that can be accessed with commands `IOCTL_CAN_READ_REGISTER` and `IOCTL_CAN_WRITE_REGISTER`. The registers are numbered 0 to `nRegisters-1`.

The `chipset_flags` entry may contain a combination of the following bit values.

`chipset_flags`	Meaning
`CANBUS_CFS_CAN_2_0_A`	Controller supports CAN2.0A
`CANBUS_CFS_CAN_2_0_B`	Controller supports CAN2.0B
`CANBUS_CFS_EXT_FRAME`	Controller supports extended frames
`CANBUS_CFS_POLLING`	Controller supports polling (default: interrupts)

8.23 IOCTL_CAN_SET_COMMAND

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_SET_COMMAND`
`lpInBuffer`	Pointer to `DWORD` with the CAN command to execute (see below)
`nInBufferSize`	`sizeof(DWORD)`
`lpOutBuffer`	Unused, set to `NULL`
`nOutBufferSize`	Unused, set to `0`
`lpReturned`	Unused, set to `NULL`
`lpOverlapped`	Unused, set to `NULL`

CAN Command in `lpInBuffer`:

Command	Description
`CANBUS_CMD_ENTER_STANDBY`	Switch CAN controller to a power-saving sleep mode
`CANBUS_CMD_LEAVE_STANDBY`	Manually leave sleep mode.
`CANBUS_CMD_LISTEN_ON`	Switch CAN controller to listen-only mode
`CANBUS_CMD_LISTEN_OFF`	Leave listen-only mode
`CANBUS_CMD_VIRTUALIZE_ON`	Use virtual CAN bus between different file handles
`CANBUS_CMD_VIRTUALIZE_OFF`	Don't use virtual CAN bus between file handles
`CANBUS_CMD_CLEAR_OVERRUN`	Clear overrun flag of the CAN controller after event `CANBUS_EVENT_OVERRUN`
`CANBUS_CMD_ABORT _TRANSMISSION`	Abort any currently active message transmissions on this virtual send channel and purge the send buffer
`CANBUS_CMD_SELF _RECEPTION_REQUEST`	Switch CAN controller to a hardware loop-back test mode where it can only receive its own messages.

Return:

0	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Issue a special low-level command. Whether a specific command is supported on a specific driver/controller combination can be determined by `IOCTL_CAN_READ_PROPERTIES`.

Standby

When entering standby mode, the CAN controller is set to a power-saving sleep mode. This is a global setting impacting all open file handles. Therefore all virtual receive channels will receive a `CANBUS_EVENT_ENTERING_STANDBY` event.
The sleep mode is either switched off manually by command `CANBUS_CMD_LEAVE_STANDBY`, or automatically by any activity on the bus (sending or receiving). Therefore calling any send function will automatically wake up the CAN controller.
When the CAN controller leaves standby mode for any reason, all active virtual receive channels are informed by event `CANBUS_EVENT_LEAVING_STANDBY`.

Listen

When activating listen-only mode, the transmitter of the CAN controller is switched off. This is useful to avoid sending anything by accident. If an application tries to send something while in listen-only mode, the messages will accumulate in the send buffer and not be transmitted until back in normal mode.
As the receiver of the CAN controller is still active in listen-only mode, the reception of messages can continue normally.
Switching listen-only mode on or off is a global setting. Therefore all active virtual receive channels will get an event `CANBUS_EVENT_DEVICE_CHANGED`.

Virtualize

It is possible to open several file handles to the CAN driver at the same time. If this is done from different applications that implement completely separate devices, these applications might assume that they can communicate with each other over the CAN bus. However for technical reasons, a CAN controller can not receive the messages it has sent itself. As a consequence these applications can not see any message sent by one of them.
This is where Virtualize will help. It activates a virtual CAN bus between these applications, not handled by the controller hardware, but by the driver itself. When one application sends a message and another application has a matching acceptance filter, the message is transported directly to the event queue of the receiving application, as if it actually had been received via the CAN bus.

Virtualize	Meaning
`OFF`	No virtual CAN bus. A message that is sent can only be received by other devices. No other open file handle on this board will receive this message, even if the acceptance filter matches.
`ON`	Use virtual CAN bus. Every other open file handle on this board with matching acceptance filter will receive a sent message as if it had actually been transmitted over the CAN bus.

This setting can be configured by registry entry `Virtualize`.

Example:

```
HANDLE hCAN;
CAN_EVENT cEvent;
CAN_DWORD dwCmd = CANBUS_CMD_CLEAR_OVERRUN;
hCAN = CreateFile(..., GENERIC_READ, ...);
DeviceIoControl(hCAN, IOCTL_CAN_READ_EVENT_DATA,
      NULL, 0, &cEvent, sizeof(CAN_EVENT), NULL, NULL);
if (cEvent.event == CANBUS_EVENT_OVERRUN)
{
      DeviceIoControl(hCAN, IOCTL_CAN_SET_COMMAND,
            &dwCmd, sizeof(dwCmd), NULL, 0, NULL, NULL);
}
```

8.24 IOCTL_CAN_READ_REGISTER

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_READ_REGISTER`
`lpInBuffer`	Pointer to a `BYTE` with the register number (address) to read
`nInBufferSize`	`sizeof(BYTE)`
`lpOutBuffer`	Pointer to a `BYTE` that receives the register value
`nOutBufferSize`	`sizeof(BYTE)`
`lpReturned`	If not `NULL`, the referenced `DWORD` will be set to the number of actually returned bytes in `lpOutBuffer`
`lpOverlapped`	Unused, set to `NULL`

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Read the value of the given CAN controller register.
Attention! Reading a register directly from the CAN controller bypasses the CAN driver and may cause unexpected and unpredictable side effects, even failure.

8.25 IOCTL_CAN_WRITE_REGISTER

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_WRITE_REGISTER`
`lpInBuffer`	Pointer to a `BYTE` with the register number (address) to write
`nInBufferSize`	`sizeof(BYTE)`
`lpOutBuffer`	Pointer to a `BYTE` that holds the new value to set
`nOutBufferSize`	`sizeof(BYTE)`
`lpReturned`	Unused, set to `NULL`
`lpOverlapped`	Unused, set to `NULL`

Return:

0	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Write a new value to the given CAN controller register.
This call uses both data pointers of `DeviceIoControl()` as `IN` pointers, `lpInBuffer` and `lpOutBuffer`. This is a little bit unusual, but works nonetheless.
Attention! Writing a value directly to the CAN controller register bypasses the CAN driver and may cause unexpected and unpredictable side effects, even failure.

8.26 IOCTL_CAN_READ_REGISTER_RM

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_READ_REGISTER_RM`
`lpInBuffer`	Pointer to a `BYTE` with the register number (address) to read
`nInBufferSize`	`sizeof(BYTE)`
`lpOutBuffer`	Pointer to a `BYTE` that receives the register value
`nOutBufferSize`	`sizeof(BYTE)`
`lpReturned`	If not `NULL`, the referenced `DWORD` will be set to the number of actually returned bytes in `lpOutBuffer`
`lpOverlapped`	Unused, set to `NULL`

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Read the value of the given CAN controller register in Configuration Mode (sometimes called Reset Mode). This command can be used to read configuration data not accessible by the normal `IOCTL_CAN_READ_REGISTER` command.
Attention! Reading a register directly from the CAN controller bypasses the CAN driver and may cause unexpected and unpredictable side effects, even failure.

8.27 IOCTL_CAN_WRITE_REGISTER_RM

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_WRITE_REGISTER_RM`
`lpInBuffer`	Pointer to a `BYTE` with the register number (address) to write
`nInBufferSize`	`sizeof(BYTE)`
`lpOutBuffer`	Pointer to a `BYTE` that holds the new value to set
`nOutBufferSize`	`sizeof(BYTE)`
`lpReturned`	Unused, set to `NULL`
`lpOverlapped`	Unused, set to `NULL`

Return:

`0`	Error, see `GetLastError()` for details
`!=0`	Success

Description:
Write a new value to the given CAN controller register in Configuration Mode (sometimes called Reset Mode). This command can be used to modify configuration data not accessible by the normal `IOCTL_CAN_WRITE_REGISTER` command..
This call uses both data pointers of `DeviceIoControl()` as `IN` pointers, `lpInBuffer` and `lpOutBuffer`. This is a little bit unusual, but works nonetheless.
Attention! Writing a value directly to the CAN controller register bypasses the CAN driver and may cause unexpected and unpredictable side effects, even failure.

8.28 IOCTL_CAN_INIT

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_INIT`
`lpInBuffer`	Unused, set to `NULL`
`nInBufferSize`	Unused, set to `0`
`lpOutBuffer`	Pointer to a `DWORD` where the error status of the initialisation will be stored; this is the same value as can be retrieved with `GetLastError()`
`nOutBufferSize`	`sizeof(DWORD)`
`lpReturned`	If not `NULL`, the referenced `DWORD` will be set to the number of actually returned bytes in `lpOutBuffer`
`lpOverlapped`	Unused, set to `NULL`

Return:

0	Error, see `GetLastError()` or value in `lpOutBuffer` for details
`!=0`	Success

Description:
Reinitialises the controller with the current settings. This might be useful when the controller experienced lots of errors and went offline.

8.29 IOCTL_CAN_TEST_DEVICE

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_CAN_TEST_DEVICE`
`lpInBuffer`	Unused, set to `NULL`
`nInBufferSize`	Unused, set to `0`
`lpOutBuffer`	Unused, set to `NULL`
`nOutBufferSize`	Unused, set to `0`
`lpReturned`	Unused, set to `NULL`
`lpOverlapped`	Unused, set to `NULL`

Return:

`0`	Controller is in Normal Mode
`1`	Controller is in Configuration Mode

Description:
Test if the CAN controller is currently in Configuration Mode or not. Configuration Mode is sometimes also called Reset Mode. This may have some relevance when directly reading from or writing to CAN controller registers.
Attention! The CAN driver may internally switch to and from Configuration Mode at any time. Therefore this information here is highly unreliable. The function only exists for backward compatibility. Simply don't use it in new applications.

9 Appendix A

9.1 canbusio.h

The file `canbusio.h` must be included in all source files that want to use the CAN bus driver interface. It defines all structures, data types and IOCTL commands used by the driver. The following listing is a slightly modified version of the file to fit better into the line length of this document.

```
/*********************************************************/
/***                                                  ***/
/***                                                  ***/
/***              C A N   D r i v e r                 ***/
/***                                                  ***/
/***                     f o r                        ***/
/***                                                  ***/
/***    N e t D C U / P i c o M O D / P i c o C O M   ***/
/***                                                  ***/
/***                                                  ***/
/*********************************************************/
/*** File:     canbusio.h                             ***/
/*** Author:   H. Froelich, H. Keller                 ***/
/*** Created:  03.04.2008                             ***/
/*** Modified: 04.06.2008 18:38:03 (HK)               ***/
/***                                                  ***/
/*** Description:                                     ***/
/*** Interface to the CANINTF driver for NetDCU,      ***/
/*** PicoMOD, PicoCOM.                                ***/
/*********************************************************/
/*** THIS CODE AND INFORMATION IS PROVIDED "AS IS"     ***/
/*** WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR ***/
/*** IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED ***/
/*** WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR  ***/
/*** A PARTICULAR PURPOSE.                            ***/
/*** Copyright (c) 2008 F&S Elektronik Systeme GmbH.   ***/
/***                    All rights reserved.          ***/
/*********************************************************/
#ifndef __CANBUSIO_H__
#define __CANBUSIO_H__
/* Some data types */
#define CAN_DWORD unsigned long
#define CAN_BYTE unsigned char
#define CAN_UINT unsigned int
#define CAN_INT int
/* Maximun size of the device name */
#define MAX_DEVICE_NAME_LENGTH   100
/* Possible CAN modes */
#define CANBUS_FORMAT_CAN_2_0_A  0
#define CANBUS_FORMAT_CAN_2_0_B  1
/* Possible chipset Flags */
#define CANBUS_CFS_CAN_2_0_A (1<<0)   /* Supports CAN 2.0 A */
#define CANBUS_CFS_CAN_2_0_B (1<<1)   /* Supports CAN 2.0 B */
#define CANBUS_CFS_EXT_FRAME (1<<2)   /* Supports Ext. Frames
                                         (only in CAN 2.0B) */
#define CANBUS_CFS_POLLING   (1<<3)   /* Supports polling mode
                                         (default interrupt) */
/* Frame format for transmissions */
#define CANBUS_TRANS_FMT_DEFAULT 0    /* Use default format */
#define CANBUS_TRANS_FMT_STD     1    /* Use Standard Frame
                                         (11-bit identifier) */
#define CANBUS_TRANS_FMT_EXT     2    /* Use Extended Frame
                                         (29-bit identifier) */
/* Possible event types */
enum t_canbus_events
{
       CANBUS_EVENT_RECEIVED = 0x001,
       CANBUS_EVENT_TRANSMITTED = 0x002,
       CANBUS_EVENT_BUS_ERROR = 0x004,
       CANBUS_EVENT_WARNING = 0x008,
       CANBUS_EVENT_LEAVING_STANDBY = 0x010,
       CANBUS_EVENT_ARBITRATION_LOST = 0x020,
       CANBUS_EVENT_OVERRUN = 0x040,
       CANBUS_EVENT_PASSIVE = 0x080,
       CANBUS_EVENT_ENTERING_STANDBY = 0x100,
       CANBUS_EVENT_DEVICE_CHANGED = 0x200,
       CANBUS_EVENT_ABORTED = 0x400, /* Since V2.x */
};
/* Possible commands */
enum t_canbus_commands
{
       CANBUS_CMD_ENTER_STANDBY = 1,
       CANBUS_CMD_ABORT_TRANSMISSION,
       CANBUS_CMD_CLEAR_OVERRUN,
       CANBUS_CMD_LEAVE_STANDBY,
       CANBUS_CMD_SELF_RECEPTION_REQUEST,
       CANBUS_CMD_LISTEN_ON,
       CANBUS_CMD_LISTEN_OFF,
       CANBUS_CMD_VIRTUALIZE_ON,
       CANBUS_CMD_VIRTUALIZE_OFF,
       /* The following must be the last list entry */
       CANBUS_CMD_LAST_ENTRY
};
/* Maximum number of entries in baud rate table */
#define CANBUS_NUMBER_OF_CONSTANT_BAUDRATES 50
/* CAN time type */
typedef struct tagCAN_TIME
{
       CAN_DWORD low;
       CAN_DWORD high;
} CAN_TIME, *PCAN_TIME;
/* Acceptance filter type */
typedef struct tagCAN_ACCEPTANCE_FILTER
{
       CAN_DWORD code;
       CAN_DWORD mask;
} CAN_ACCEPTANCE_FILTER, *PCAN_ACCEPTANCE_FILTER;
/* Properties of the selected channel */
typedef struct tagCAN_PROPERTIES
{
       CAN_DWORD version;                      /* Driver version */
       TCHAR device_name[MAX_DEVICE_NAME_LENGTH];
                                               /* Name of the device */
       CAN_DWORD min;                          /* Minimum baud rate */
       CAN_DWORD max;                          /* Maximum baud rate */
       CAN_INT nCommands;                      /* Number of entries
                                                  in commands[] */
       CAN_INT commands[CANBUS_CMD_LAST_ENTRY];
                                               /* Supported commands */
       CAN_INT nBaudrates;                     /* Number of entries
                                                  in baudrates[] */
       CAN_DWORD baudrates[CANBUS_NUMBER_OF_CONSTANT_BAUDRATES];
                                               /* Preset baud rates */
       CAN_DWORD chipset_flags;                /* Available modes and
                                                  frame formats */
       CAN_INT nRegisters;                     /* Number of available
                                                  hardware registers */
} CAN_PROPERTIES, *PCAN_PROPERTIES;
/* Transmission data type; also used in event data */
typedef struct tagCAN_TRANSMIT_DATA
{
       CAN_UINT fmt;                           /* Frame format */
       CAN_DWORD identifier;                   /* CAN ID (11/29 bits) */
       CAN_BYTE rtr;                           /* Remote Transmission
                                                  Request */
       CAN_BYTE dlc;                           /* Data Length Code */
       CAN_BYTE msg[8];                        /* Data bytes */
} CAN_TRANSMIT_DATA, *PCAN_TRANSMIT_DATA;
/* Event data type; ignore fmt in entry "data" */
typedef struct tagCAN_EVENT
{
       CAN_DWORD event;                        /* Event type */
       CAN_TIME time;                          /* Time when event
                                                  occured */
       CAN_DWORD lost;                         /* Count of lost events
                                                  since last recorded */
       CAN_TRANSMIT_DATA data;                 /* Message data (depends
                                                  on event type) */
       CAN_UINT arbitration;                   /* (unused) */
} CAN_EVENT, *PCAN_EVENT;
```

```
/* We'll need some defines */
#include "WINIOCTL.h"
/* Additonal IOCTL values for CAN access */
#define CAN_DEV   0x00008007
#define CAN_BUFF  METHOD_BUFFERED
#define CAN_READ  FILE_READ_ACCESS
#define CAN_WRITE FILE_WRITE_ACCESS
#define CAN_ANY   FILE_ANY_ACCESS
#define IOCTL_CAN_WRITE_ACCEPTANCE_FILTER \
       CTL_CODE(CAN_DEV, 0x801, CAN_BUFF, CAN_WRITE)
#define IOCTL_CAN_READ_ACCEPTANCE_FILTER \
       CTL_CODE(CAN_DEV, 0x802, CAN_BUFF, CAN_READ)
#define IOCTL_CAN_SET_BAUDRATE \
       CTL_CODE(CAN_DEV, 0x803, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_GET_BAUDRATE \
       CTL_CODE(CAN_DEV, 0x804, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_INIT \
       CTL_CODE(CAN_DEV, 0x805, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_SET_BAUDRATE_BY_CONSTANT \
       CTL_CODE(CAN_DEV, 0x806, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_GET_BAUDRATE_BY_CONSTANT \
       CTL_CODE(CAN_DEV, 0x807, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_SET_CAN_MODE \
       CTL_CODE(CAN_DEV, 0x808, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_SET_COMMAND \
       CTL_CODE(CAN_DEV, 0x809, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_WRITE_TRANSMIT_DATA \
       CTL_CODE(CAN_DEV, 0x80A, CAN_BUFF, CAN_WRITE)
#define IOCTL_CAN_READ_EVENT_DATA \
       CTL_CODE(CAN_DEV, 0x80B, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_READ_TIME \
       CTL_CODE(CAN_DEV, 0x80C, CAN_BUFF, CAN_READ)
#define IOCTL_CAN_SET_DEFAULT_FRAME_FORMAT \
       CTL_CODE(CAN_DEV, 0x80D, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_TEST_DEVICE \
       CTL_CODE(CAN_DEV, 0x80E, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_READ_PROPERTIES \
       CTL_CODE(CAN_DEV, 0x80F, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_READ_REGISTER \
       CTL_CODE(CAN_DEV, 0x810, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_WRITE_REGISTER \
       CTL_CODE(CAN_DEV, 0x811, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_READ_REGISTER_RM \
       CTL_CODE(CAN_DEV, 0x812, CAN_BUFF, CAN_ANY)
#define IOCTL_CAN_WRITE_REGISTER_RM \
       CTL_CODE(CAN_DEV, 0x813, CAN_BUFF, CAN_ANY)
#endif /*!__CANBUSIO_H__*/
```

10 Appendix B

10.1 CanTestSuite

We provide a group of sample applications with the CAN bus that should show the usage and allow an easy configuration and test of the CAN bus settings and operation.

Program	Description
`CanWrite`	Command line tool that writes 1000 messages with increasing IDs to the CAN bus. Uses `WriteFile()`.
`CanRead`	Command line tool to read and log all events on the CAN bus. Uses `ReadFile()`.
`CanSend`	Command line tool to send any number of messages on any number of send channels with any number of parallel threads to the CAN bus. This can be used to stress test the driver. Uses `IOCTL_CAN_WRITE_TRANSMIT_DATA`.
`CanMon`	Command line tool to monitor all events on the CAN bus and log in an own format. Uses `IOCTL_CAN_READ_EVENT_DATA`.
`CanCheck`	More complex, dialog oriented tool to check and set the CAN bus settings. It is also possible to send test messages and monitor the CAN bus.

Table 12: Programs in CanTestSuite

We will provide the listings to `CanWrite` and `CanRead`, as they are rather short and can serve as additional examples to show how to use the CAN driver interface.

10.2 CanWrite

The `CanWrite` program shows how standard C file handles and `fprintf()` can be used to send messages to the CAN bus. Please note that this is the ASCII version of `fprintf()`, as `WriteFile()`, which is called internally by `fprintf()`, needs ASCII text, not Unicode text.
On the other hand, this program also shows the limitation that no IOCTL commands can be issued via this standard C file interface. Hence we have to open the file first with a normal Windows `HANDLE` to set the baud rate. However we don't need any READ or WRITE access to do this.

```
/*********************************************************/
/***                                                  ***/
/***        C A N   M e s s a g e   W r i t e r       ***/
/***                                                  ***/
/*********************************************************/
/*** File:      CanWrite.c                            ***/
/*** Author:    H. Keller                             ***/
/***                                                  ***/
/*** Description:                                     ***/
/*** Command line tool to send messages to a CAN bus. ***/
/*********************************************************/
/*** THIS CODE AND INFORMATION IS PROVIDED "AS IS"    ***/
/*** WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED   ***/
/*** OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE     ***/
/*** IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR     ***/
/*** FITNESS FOR A PARTICULAR PURPOSE.                ***/
/*** Copyright (c) 2008 F&S Elektronik Systeme GmbH.  ***/
/***                   All rights reserved.           ***/
/*********************************************************/
#include <windows.h>
#include "canbusio.h"
/********************************************************
*** Function:    int _tmain(int argc, TCHAR* argv[])  ***
***                                                   ***
*** Parameters:  argc: Command line argument count    ***
***              argv: Array of argument strings      ***
***                                                   ***
*** Return:      0: Success                           ***
***              1: Error, device not available       ***
***              2: Usage (e.g. after wrong argument) ***
***                                                   ***
*** Description                                       ***
*** -----------                                       ***
*** Send 1000 messages to the CAN port.               ***
********************************************************/
int _tmain(int argc, TCHAR *argv[])
{
       int i;
       TCHAR *pDevice = _T("CID1:");
       TCHAR *pBaudrate = NULL;
       TCHAR *pLogfile = NULL;
       BOOL bHelp = FALSE;
       DWORD dwBaudrate;
       HANDLE hCAN;
       FILE *pfCAN;
       FILE *pfLogfile;

       /* Say hello */
       _tprintf(_T("CanWrite V3.0\n"));

       /* Parse command line arguments */
       for (i=1; i<argc; i++)
       {
              TCHAR *p = argv[i];
              /* Check for options */
              if (*p != '-')
              {
                     _tprintf(_T("Unknown argument '%s'\n"), p);
                     bHelp = TRUE;
                     break;
              }
              p++;
```

```
                /* Parse options without an argument */
                if ((*p == '?') || (*p == 'h'))
                {
                        bHelp = TRUE;
                        break;
                }
                /* Remaining options need an argument */
                if (i+1 >= argc)
                {
                        _tprintf(_T("Missing argument to option ")
                                 _T("-%c\n"), *p);
                        bHelp = TRUE;
                        break;
                }
                else
                {
                        i++;
                        if (*p == 'd')
                                pDevice = argv[i];
                        else if (*p == 'b')
                                pBaudrate = argv[i];
                        else if (*p == 'f')
                                pLogfile = argv[i];
                        else
                        {
                                _tprintf(_T("Unknown option -%c\n"), *p);
                                bHelp = TRUE;
                                break;
                        }
                }
        }

        /* Print usage, if required */
        if (bHelp)
        {
                _tprintf(_T("CanWrite\n")
                         _T("  -d <dev> : Device name (CID1:)\n")
                         _T("  -b <baud>: Set baudrate\n")
                         _T("  -f <name>: Log file name\n")
                         _T("  -h       : Show this help\n"));
                return 2;
        }

        /* Open device in configuration mode (no access) */
        hCAN = CreateFile(pDevice, 0, 0, NULL,
                          OPEN_EXISTING, 0, NULL);
        if( hCAN == INVALID_HANDLE_VALUE )
        {
                _tprintf(_T("Can't open device %s, error %d\n"),
                         pDevice, GetLastError());
                return 1;
        }

        /* If requested, set new baud rate */
        if (pBaudrate)
        {
                dwBaudrate = _tcstoul(pBaudrate, NULL, 0);
                DeviceIoControl(hCAN, IOCTL_CAN_SET_BAUDRATE,
                                &dwBaudrate, sizeof(dwBaudrate),
                                NULL, 0, NULL, NULL);
        }

        /* Read current baud rate */
        DeviceIoControl(hCAN, IOCTL_CAN_GET_BAUDRATE, NULL,
                        0, &dwBaudrate, sizeof(dwBaudrate),
                        NULL, NULL);
        /* Close file again */
        CloseHandle(hCAN);

        /* Open device with standard file handle */
        pfCAN = _wfopen(pDevice, _T("w+t"));
        if (pfCAN)
        {
                /* Open log file */
                if (pLogfile)
                {
                        pfLogfile = _wfopen(pLogfile, _T("w"));
                        if (!pfLogfile)
                        {
                                _tprintf(_T("Can't open log file '%s', ")
                                         _T("error %d\n"), pLogfile,
                                         GetLastError());
                                pLogfile = NULL;
                        }
                        else
                                _tprintf(_T("Using log file '%s'\n"),
                                         pLogfile);
                }
                /* Send 1000 messages */
                _tprintf(_T("Start sending 1000 messages to %s, ")
                         _T("baudrate=%d Hz\n"), pDevice,
                         dwBaudrate);
                for (i=0; i<1000; i++)
                {
                        /* Send one message to the CAN port */
                        fprintf(pfCAN, "%x 0 8 0 0 0 0 0 0 0 0\n", i);
                        fflush(pfCAN);
                        /* Write the same message to the log file */
                        if (pLogfile)
                        {
                                fprintf(pfLogfile,
                                        "%x 0 8 0 0 0 0 0 0 0 0\n", i);
                                fflush(pfLogfile);
                        }
                }
                /* Close log file */
                if (pLogfile)
                        fclose(pfLogfile);
                /* Close CAN device */
                fclose(pfCAN);
        }
        return 0;
}
```

10.3 CanRead

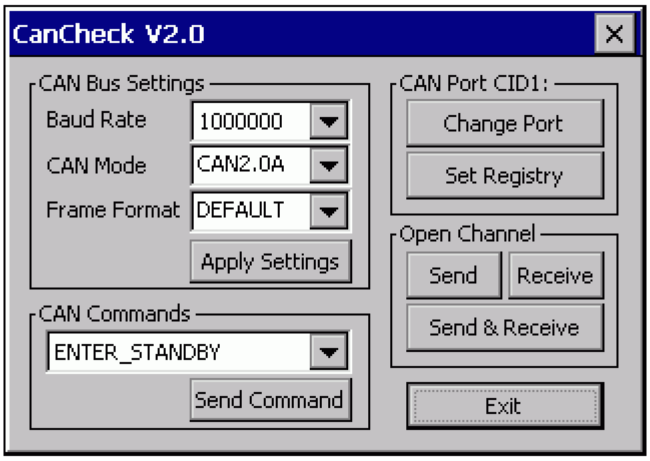
The `CanRead` program shows how the text interface via `ReadFile()` can be used to directly output the arriving event without any further processing. Please note how we use `printf()` instead of `_tprintf()` in the appropriate line when writing the buffer to `stdout`, as the result from `ReadFile()` is ASCII text, not Unicode text.

```
/********************************************************/
/***                                                  ***/
/***          C A N   E v e n t   R e a d e r         ***/
/***                                                  ***/
/********************************************************/
/*** File:     CanRead.c                              ***/
/*** Author:   H. Keller                              ***/
/***                                                  ***/
/*** Description:                                     ***/
/*** Command line tool to read events from a CAN bus. ***/
/********************************************************/
/*** THIS CODE AND INFORMATION IS PROVIDED "AS IS"    ***/
/*** WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED   ***/
/*** OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE     ***/
/*** IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR     ***/
/*** FITNESS FOR A PARTICULAR PURPOSE.                ***/
/*** Copyright (c) 2008 F&S Elektronik Systeme GmbH.  ***/
/***                   All rights reserved.           ***/
/********************************************************/
#include <windows.h>
#include "canbusio.h"

/********************************************************
*** Function:    int _tmain(int argc, TCHAR* argv[])  ***
***                                                   ***
*** Parameters:  argc: Command line argument count    ***
***              argv: Array of argument strings      ***
***                                                   ***
*** Return:      0: Success                           ***
***              1: Error, device not available       ***
***              2: Usage (e.g. after wrong argument) ***
***                                                   ***
*** Description                                       ***
*** -----------                                       ***
*** Monitor all events coming in on the CAN port      ***
********************************************************/
int _tmain(int argc, TCHAR *argv[])
{
        int i;
        TCHAR *pDevice = _T("CID1:");
        TCHAR *pBaudrate = NULL;
        TCHAR *pLogfile = NULL;
        BOOL bHelp = FALSE;
        DWORD dwBaudrate;
        HANDLE hLogfile;
        HANDLE hCAN;
        DWORD dwRead;
        char pBuffer[100];
        CAN_ACCEPTANCE_FILTER cFilter;
        COMMTIMEOUTS cTimeouts;

        /* Say hello */
        _tprintf(_T("CanRead V3.0\n"));

□       /* Parse command line arguments */
        for (i=1; i<argc; i++)
        {
                TCHAR *p = argv[i];

□               /* Check for options */
                if (*p != '-')
                {
                        _tprintf(_T("Unknown argument '%s'\n"), p);
                        bHelp = TRUE;
                        break;
                }

                /* Parse options without an argument */
                if ((*p == '?') || (*p == 'h'))
                {
                        bHelp = TRUE;
                        break;
                }
        /* Parse command line arguments */
        for (i=1; i<argc; i++)
        {
                TCHAR *p = argv[i];

                /* Check for options */
                if (*p != '-')
                {
                        _tprintf(_T("Unknown argument '%s'\n"), p);
                        bHelp = TRUE;
                        break;
                }

                /* Parse options without an argument */
                if ((*p == '?') || (*p == 'h'))
                {
                        bHelp = TRUE;
                        break;
                }
                /* Check for options */
                if (*p != '-')
                {
                        _tprintf(_T("Unknown argument '%s'\n"), p);
                        bHelp = TRUE;
                        break;
                }

                /* Parse options without an argument */
                if ((*p == '?') || (*p == 'h'))
                {
                        bHelp = TRUE;
                        break;
                }
                /* Remaining options need an argument */
                if (i+1 >= argc)
                {
                        _tprintf(_T("Missing argument to option ")
                                 _T("-%c\n"), *p);
                        bHelp = TRUE;
                        break;
                }
                else
                {
                        i++;
                        if (*p == 'd')
                                pDevice = argv[i];
                        else if (*p == 'b')
                                pBaudrate = argv[i];
                        else if (*p == 'f')
                                pLogfile = argv[i];
                        else
                        {
                                _tprintf(_T("Unknown option -%c\n"), *p);
                                Help = TRUE;
                                break;
                        }
                }
        }
        /* Print usage, if required */
        if (bHelp)
        {
                _tprintf(_T("CanRead\n")
                         _T("   -d <dev> : Device name (CID1:)\n")
                         _T("   -b <baud>: Set baudrate\n")
                         _T("   -f <name>: Log file name\n")
                         _T("   -h       : Show this help\n"));
                return 2;
        }

        /* Open device */
        hCAN = CreateFile(pDevice, GENERIC_READ, 0, NULL,
                          OPEN_EXISTING, 0, NULL);
        if (hCAN == INVALID_HANDLE_VALUE)
        {
                _tprintf(_T("Can't open device %s, error %d\n"),
                          pDevice, GetLastError());
                return 1;
        }

        /* If requested, set new baud rate */
        if (pBaudrate)
        {
                dwBaudrate = _tcstoul(pBaudrate, NULL, 0);
                DeviceIoControl(hCAN, IOCTL_CAN_SET_BAUDRATE,
                                 &dwBaudrate, sizeof(dwBaudrate),
                                 NULL, 0, NULL, NULL);
        }

        /* Read current baud rate */
        DeviceIoControl(hCAN, IOCTL_CAN_GET_BAUDRATE, NULL,
                          0, &dwBaudrate, sizeof(dwBaudrate),
                          NULL, NULL);

        /* Set acceptance filter to accept all IDs */
        cFilter.mask = 0xFFFFFFFF;
        cFilter.code = 0x00000000;
        DeviceIoControl(hCAN, IOCTL_CAN_WRITE_ACCEPTANCE_FILTER,
                         &cFilter, sizeof(cFilter), NULL, 0,
                         NULL, NULL);

        /* Set read timeout to infinite */
        cTimeouts.ReadTotalTimeoutConstant = INFINITE;
        SetCommTimeouts(hCAN, &cTimeouts);

        /* Open log file */
        if (pLogfile)
        {
                hLogfile = CreateFile(pLogfile, GENERIC_WRITE, 0,
                                       NULL, CREATE_ALWAYS,
                                       FILE_FLAG_WRITE_THROUGH,
                                       NULL);
                if (hLogfile == INVALID_HANDLE_VALUE)
                {
                        _tprintf(_T("Can't open log file '%s', ")
                                  _T("error %d\n"), pLogfile,
                                  GetLastError());
                        pLogfile = NULL;
                }
                else
                        _tprintf(_T("Using log file '%s'\n"),
                                  pLogfile);
        }

        /* Start receiving events */
        _tprintf(_T("Start receiving events from %s, ")
                  _T("baudrate=%d Hz\n"), pDevice, dwBaudrate);
        for (;;)
        {
                /* Check for Ctl-C */
                if (GetAsyncKeyState(VK_CONTROL)
                    && GetAsyncKeyState('C'))
                        break;
                /* Read next event */
                if (!ReadFile(hCAN, pBuffer, sizeof(pBuffer),
                              &dwRead, NULL))
                {
                        _tprintf(_T("Read error %d\n"),
                                  GetLastError());
                        break;
                }
                if (dwRead)
                {
                        /* Print event */
                        printf(pBuffer);

                        /* Write event to log file */
                        if (pLogfile)
                        {
                                DWORD dwWritten;

                                WriteFile(hLogfile, pBuffer, dwRead,
                                             &dwWritten, NULL);
                         }
                 }
         }

         /* Close log file */
         if (pLogfile)
                 CloseHandle(hLogfile);

         /* Close CAN device */
         CloseHandle(hCAN);
         return 0;
}
```

10.4 CanCheck

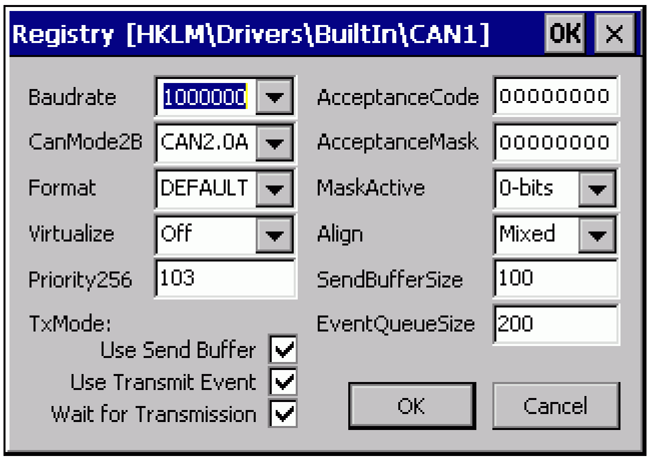
The CanCheck program provides a graphical frontend to comfortably configure and test the CAN port. It allows to send messages by creating message generators, reports all received events, and allows to set acceptance filter, event mask, baud rate, CAN mode, frame format, and last but not least all registry settings.

Main Dialog
If you start the program, you see the following dialog.



In the section *CAN Bus Settings* you can set the baud rate, the CAN mode, and the frame format by clicking on the triangle and selecting one of the entries that appear in the drop-down list. The baud rate can also be entered directly as a number.
In section *CAN Commands* you can send a special command to the current CAN port.
In section *CAN Port CIDx:* you see which CAN port currently is in use and you can switch to another port, if the board supports more than one CAN port. You can also open a new dialog where you can modify all registry settings.
Section *Open Channel* finally allows to open virtual send and receive channels. You can either do this independently of each other, or you can open a combined send and receive channel. The difference is that you'll only get events of type `CANBUS_EVENT_TRANSMITTED` if you open a combined channel (see description of virtual channels on page 22).
Each click on one of these buttons will open a separate window with the specific send or receive information. When clicking on *Send & Receive*, both types of windows will open at once. You can see from the file handle that will be displayed in the title bar of these windows that they belong together.

Registry Settings Dialog
If you clicked on button *Set Registry* in the main dialog, a new dialog shows up.


Registry [HKLM\Drivers\BuiltIn\CAN1]

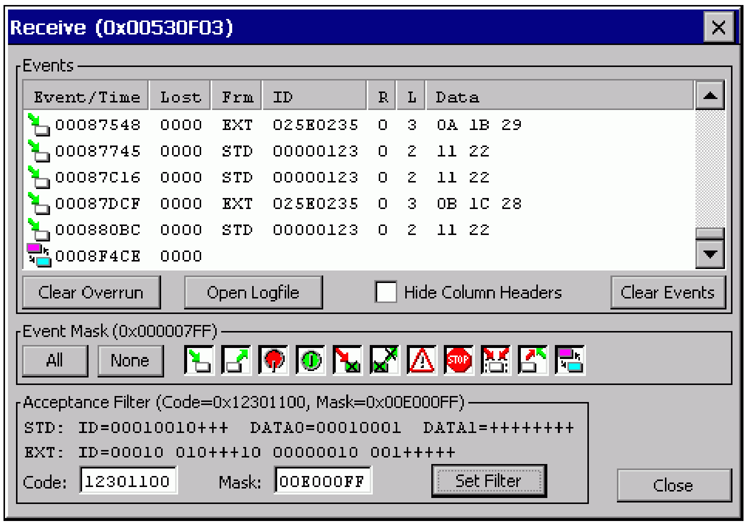
Baudrate	1000000	AcceptanceCode	00000000
CanMode2B	CAN2.0A	AcceptanceMask	00000000
Format	DEFAULT	MaskActive	0-bits
Virtualize	Off	Align	Mixed
Priority256	103	SendBufferSize	100
		EventQueueSize	200

TxMode:
Use Send Buffer ✓
Use Transmit Event ✓
Wait for Transmission ✓

Here again you can select values from the drop-down lists, or you can enter the numbers directly into the fields. Please note that `AcceptanceCode` and `AcceptanceMask` are represented as hexadecimal numbers. For the meaning of these values, please refer to the detailed description of registry values on page 15.
If you close the dialog with *OK*, the new values are directly written into the registry. Otherwise with *Cancel*, the changes are discarded and the registry remains unchanged.
Please note: As new registry values only get active after restarting the board, the program will still use the previous values further on, even if you acknowledged the dialog with *OK*. This lasts until you leave the program.

Receive Dialog

If you opened a receive channel window by pressing one of the buttons *Receive* or *Send & Receive* in the main dialog, you will be presented with the following view.



```
Receive (0x00530F03)

Events
Event/Time  Lost  Frm  ID        R  L  Data
00087548    0000  EXT  025E0235  0  3  0A 1B 29
00087745    0000  STD  00000123  0  2  11 22
00087C16    0000  STD  00000123  0  2  11 22
00087DCF    0000  EXT  025E0235  0  3  0B 1C 28
000880BC    0000  STD  00000123  0  2  11 22
0008F4CE    0000

Acceptance Filter (Code=0x12301100, Mask=0x00E000FF)
STD: ID=00010010+++  DATA0=00010001  DATA1=++++++++
EXT: ID=00010 010+++10 00000010 001+++++
Code: 12301100      Mask: 00E000FF
```

In section *Events* you'll see a list of events that occurred on this virtual channel. From left to right you see the following entries.

- Column *Event/Time* shows the event type as an icon and the time when this event happened. In fact this is only the low part of the `CAN_TIME` structure.
- Column *Lost* shows the number of lost events since the previous entry.

In case of a received, transmitted or aborted message, there are additional entries.

- Column *Frm* shows the frame type of the message: *STD* for a Standard Frame or *EXT* for an Extended Frame.
- Column *ID* shows the message ID as given in the `CAN_TRANSMIT_DATA` structure, i.e. aligned as configured with registry value `Align`.
- Column *R* shows whether the message had the RTR flag set (1) or not (0).
- Column *L* shows the data length code DLC (0 to F).
- Finally column *Data* shows zero to eight data bytes.

All values are given hexadecimal!

The event types are shown as an icon. They have the following meaning.

Icon	Event type
↘	`CANBUS_EVENT_RECEIVED`
↗	`CANBUS_EVENT_TRANSMITTED`
●	`CANBUS_EVENT_ENTERING_STANDBY`
●	`CANBUS_EVENT_LEAVING_STANDBY`
↘	`CANBUS_EVENT_OVERRUN`
↗	`CANBUS_EVENT_ABORTED`
⚠	`CANBUS_EVENT_WARNING`
STOP	`CANBUS_EVENT_BUS_ERROR`
✕	`CANBUS_EVENT_PASSIVE`
↗	`CANBUS_EVENT_ARBITRATION_LOST`
⇄	`CANBUS_EVENT_CHANGED`

If you press button *Overrun*, `CANBUS_CMD_CLEAR_OVERRUN` is sent to the CAN bus. This can be used to clear an overrun status.
Button *Open Logfile* is used to start writing the event list also to a file. You will be presented a file selection dialog to enter a file name. This file will be opened and all subsequent events are not only shown in the window, but also written to the file as text. The name of the log file will be shown in the group frame of this section. At the same time, button *Open Logfile* changes to *Close Logfile*. Clicking on it now will stop logging to the file.
Button *Hide Column Headers* will remove the header line from the event list. This makes room for another row of event data and may be useful on very small displays where only very few rows are visible.
Finally button Clear Events will simply erase the event list. This has no influence on any currently open logfile.

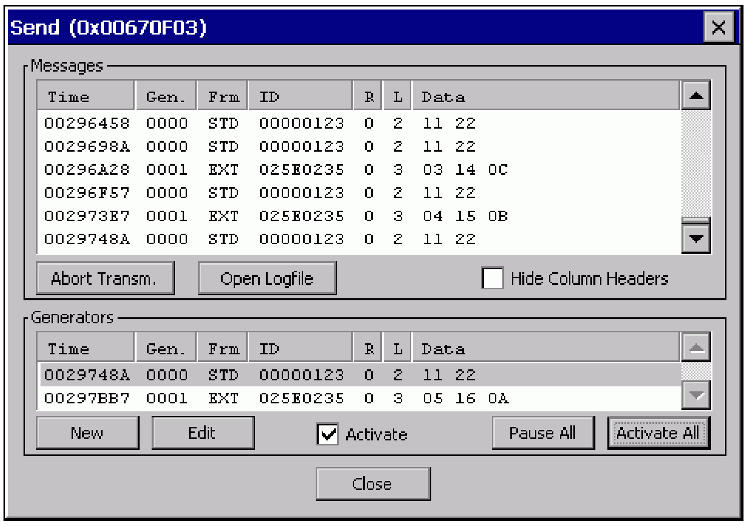
In section *Event Mask* you can tell which events should be recorded in the event list. An event type that is active has a pushed down button, and event that is not active is shown as unpushed button. You can either toggle a single event type by simply clicking on the icon button, or you can activate or deactivate all event types at once by clicking on the *All* or the *None* button. The change takes effect immediately. The current setting in hexadecimal is given in the group frame description itself.
In section *Acceptance Filter* you can modify the filter settings for this virtual channel. The current settings can be seen in hexadecimal in the frame description. The row *STD* shows how the filter will work when receiving Standard Frames, and the row *EXT* shows the filter when receiving Extended Frames. These meanings depend on the current registry settings of the `Align` and `MaskActive` values. The possible bit values are:

Bit value	Meaning
0	Bit must be zero to be accepted
1	Bit must be one to be accepted
+	Bit is always accepted

A new filter value can be entered in the white edit fields at the bottom as two hexadecimal values for *Code* and *Mask*. The change takes effect after you press button *Set Filter*.

Send Dialog

If you opened a send channel window by pressing one of the buttons *Send* or *Send & Receive* in the main dialog, you will be presented with the following view.


Send (0x00670F03)

Messages

Time	Gen.	Frm	ID	R	L	Data
00296458	0000	STD	00000123	0	2	11 22
0029698A	0000	STD	00000123	0	2	11 22
00296A28	0001	EXT	025E0235	0	3	03 14 0C
00296F57	0000	STD	00000123	0	2	11 22
002973E7	0001	EXT	025E0235	0	3	04 15 0B
0029748A	0000	STD	00000123	0	2	11 22

Generators

Time	Gen.	Frm	ID	R	L	Data
0029748A	0000	STD	00000123	0	2	11 22
00297BB7	0001	EXT	025E0235	0	3	05 16 0A

The idea behind this window is to create so called message generators that automatically generate messages with predefined content and in predefined time intervals. These generators can also automatically increment or decrement the values they send and they can repeat or only run for one cycle. Any number of generators can be created.

Section *Messages* shows a list of all actually sent messages. Each row represents one message. From left to right you see the following entries.

- Column *Time* shows the time when this message was sent. In fact this is only the low part of the `CAN_TIME` structure.
- Column *Gen.* shows the index number of the message generator that created this message. Generator indexes are counted beginning with zero.
- Column *Frm* shows the frame type of the message: *STD* for a Standard Frame or *EXT* for an Extended Frame.
- Column *ID* shows the message ID as given in the `CAN_TRANSMIT_DATA` structure, i.e. aligned as configured with registry value `Align`.
- Column *R* shows whether the message had the RTR flag set (1) or not (0).
- Column *L* shows the data length code DLC (0 to F).
- Finally column *Data* shows zero to eight data bytes.

All values are given hexadecimal!

If you press button *Abort Transmission*, the command `CANBUS_CMD_ABORT_TRANSMISSION` is sent to the CAN bus. This will abort the current transmission and will also clear the send buffer.

Button *Open Logfile* is used to start writing the message list also to a file. You will be presented a file selection dialog to enter a file name. This file will be opened and all subsequent messages are not only shown in the window, but also written to this file as text. The name of the log file will be shown in the group frame of this section. At the same time, button *Open Logfile* changes to *Close Logfile*. Clicking on it now will stop logging to the file.

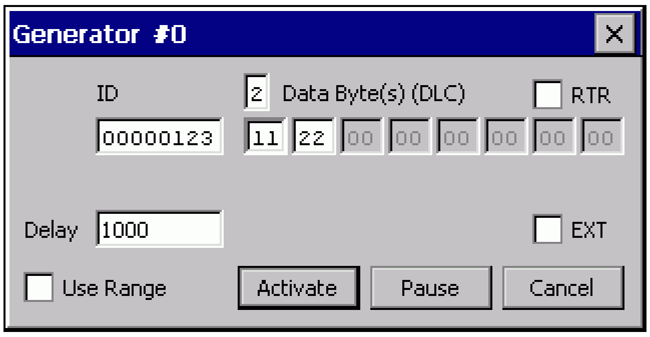
Button *Hide Column Headers* will remove the header line from the message list and the generators list (see below). This makes room for another row of data in each of these lists and may be useful on very small displays where only very few rows are visible.

Section *Generators* shows a list of all available message generators. To be precise, each row shows the time, when the generator will send its next message and the message itself, that it will send. The meaning of the columns is the same as in the messages list above. Column *Gen.* shows the index number of this generator.

At the beginning, this list is empty. You can add new generators by clicking on button *New* and entering the appropriate generator information (see below).
A message generator may be active or paused. An active generator will generate a new message after some time, a paused generator will not generate new messages. You can select any generator in the list and toggle check mark *Activate* to switch between active and paused state. Or you can pause or activate all generators at once by clicking on button *Pause All* or *Activate All* respectively. A paused generator will show

```
-Paused-
```

instead of a new time.
By clicking on button *Edit*, you can modify the settings of the currently selected message generator and by clicking on button *New*, you can add a new message generator. In both cases you are presented with a new window.


Generator #0

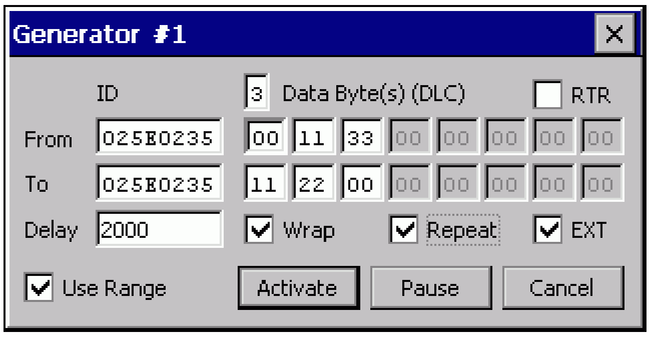
ID: 00000123
Data Byte(s) (DLC): 2
RTR: [ ]
Data bytes: 11 22 00 00 00 00 00 00
Delay: 1000
EXT: [ ]
Use Range: [ ]

Here you can enter all data of the generator.

- Field *ID* allows to enter the message ID that will be used when sending the message. It will be aligned as configured with registry value `Align`.
- Field *Data Byte(s) (DLC)* allows to enter a number between 0 and 8. It is the data length code DLC.
- Depending on the number in DLC, the according number of data byte fields below will be enabled and you can enter the appropriate values there.
- If you want to set the RTR flag of the message, check the field *RTR*. Each click toggles the check mark.
- If you want to send a message in Extended Frame Format, check the field EXT. Otherwise a message with Standard Frame Format will be sent. Each click toggles the check mark.
  Remark: This field is only available if the CAN bus is in CAN2.0B mode.
- Field *Delay* defines the time in milliseconds, before the message repeats.

All numbers but *Delay* are given hexadecimal, *Delay* is given decimal.

There exists a second form of generator, the so called Range Generator. By checking field *Use Range*, the dialog changes to the following view.


Generator #1

ID
Data Byte(s) (DLC): 3
RTR: [ ]
From: 025E0235 — 00 11 33 00 00 00 00 00
To: 025E0235 — 11 22 00 00 00 00 00 00
Delay: 2000
Wrap: [✓]
Repeat: [✓]
EXT: [✓]
Use Range: [✓]

The difference here is, that you can give *From* and *To* values for ID and data bytes. This means the generator will start with the given values from line *From* and send this as first message. Then it will increase (or decrease) the values by one and after the given *Delay*, it will send this new message. If the *From* value is smaller than the *To* value, the value is incremented each time. If the *From* value is higher than the *To* value, the value is decremented each time.

The two fields *Wrap* and *Repeat* that appear when *Use Range* is activated, decide what happens when the *To* value is reached. If *Wrap* is active, each value that reached *To* will start again at *From*. Otherwise it will stay at the *To* value. If *Repeat* is active, the generator will run indefinitely. If *Repeat* is not active, the generator automatically will switch to state `Paused` if it has completed one cycle (counted from the largest value difference in any of the values). It can be re-activated for another cycle then.

Example 1
The message generator shown in the window above will generate an Extended Frame message with a fix ID `0x025E0235` and three data bytes every two seconds.
The first data byte will be `0x00` in the first cycle, `0x01` in the second cycle and so on until it reaches `0x11`. Then it will wrap around back to `0x00` again in the next cycle.
The second data byte will be `0x11` in the first cycle, `0x12` in the second cycle and so on until it reaches `0x22`. Then it will wrap around back to `0x11` in the next cycle.
The third data byte will be `0x33` in the first cycle, `0x32` in the second cycle and so on until it reaches `0x00`. Then it will wrap around back to `0x33` in the next cycle.
The generator does not stop automatically because *Repeat* is activated.

Example 2
A generator with the same values, but *Repeat* not activated, would stop after 0x34 = 52 messages, as `0x33` to `0x00` is the largest difference of values. That means it stops when the third data byte has reached its *To* value of `0x00`. The first two data bytes would have gone three times through their whole range during this time.

To close the message generator dialog, you can press on one of the buttons *Activate*, *Pause*, or *Cancel*. Button *Activate* takes the new settings and the generator is immediately activated, i.e. sends the first message right away. Button *Pause* takes the settings, but the generator is paused, i.e. does not generate a message until activated later. And when pressing button *Cancel*, the new settings are discarded. That means either no new generator is created or, if you were editing an existing generator, this generator remains unchanged.

Remark
When editing an existing generator, this generator is automatically paused while in the generator dialog. If you return by *Cancel*, the previous state is resumed. Otherwise the new settings are taken instead and the new state depends on whether you have pressed *Activate* or *Pause*.

11 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 authorised 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 unauthorised 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.

12 Warranty Terms

Hardware Warranties
F&S guarantees hardware products against defects in workmanship and material for a period of two (2) years 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

Tables